Hardware-assisted Occlusion Culling for Scene Graph Systems

Dissertation
der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. Dirk Staneker
aus Reutlingen

Tübingen
2005

Tag der mündlichen Qualifikation:	25.01.2006
Dekan:	Prof. Dr. Michael Diehl
1. Berichterstatter:	Prof. Dr.-Ing. Dr.-Ing. E.h. Wolfgang Straßer
2. Berichterstatter:	Prof. Dr. Andreas Schilling

Abstract

This thesis describes different aspects of occlusion culling algorithms for the efficient rendering of 3D scenes with rasterization hardware. Scene graphs are used as data structures for the scenes to support a wide range of different applications. All presented algorithms permit modifications of the graphs at runtime and therefore the algorithms are suitable for dynamic scenes.

The thesis consists of three parts. The first part compares different techniques to determine the visibility of a single scene graph node. All techniques have the same characteristic in that they utilize the depth information on the graphics hardware in some way. Unfortunately, each access to the hardware requires some latency. Therefore the second part of this thesis presents some algorithms to reduce the number of these accesses to the graphics hardware. The algorithms take advantage of a lower resolution representation of the scene graph nodes in screen-space on the one hand, and they also use the informations of previous occlusion queries on the other. Because all the presented algorithms use the depth values of the currently rendered scene, the order of the rendering and the occlusion tests is important. Hence the third part of this thesis presents a novel algorithm for the traversal of a scene graph which efficiently utilizes hardware occlusion queries. Therefore the algorithm uses screen-space coherence in combination with a front-to-back sorted traversal of the scene graph in object-space. To determine the occlusion, the algorithm bundles individual occlusion tests in multiple occlusion queries. These can be asynchronously performed to reduce the latency.

All presented algorithms deliberately do not use special – spatial – data structures for the scene to avoid long preprocessing times or restrictions in the use of dynamic scenes. Also, the algorithms do not exploit temporal coherence between the frames, because this results in limitations for dynamic and interactive scenes. However, the presented algorithms admit an efficient rendering of scenes with high depth complexity.

Zusammenfassung

Die vorliegende Arbeit behandelt verschiedene Aspekte der Verdeckungsrechnung zur effizienteren Darstellung dreidimensionaler Szenen mit Hilfe von Rasterisierungshardware. Alle betrachteten Algorithmen verwenden einen herkömmlichen Szenegraphen als grundlegende Datenstruktur. Dadurch lassen sie sich unmittelbar zahlreichen Anwendungen zur Verfügung stellen. Alle Algorithmen erlauben es, den Szenegraphen zur Laufzeit zu modifizieren, und lassen sich somit auch auf dynamische Szenen anwenden.

Die Arbeit teilt sich auf in drei Bereiche. Im ersten Abschnitt wird auf die verschiedenen Möglichkeiten eingegangen, die Verdeckung eines einzelnen Knotens aus dem Szenegraphen zu ermitteln. Die vorgestellten Algorithmen implementieren dabei verschiedene Methoden, um die auf der Graphikhardware gespeicherte Tiefeninformation auszunutzen. Allerdings benötigt jeder Zugriff auf die Graphikhardware eine kurze Zeitspanne. Der zweite Abschnitt beschäftigt sich daher damit, Zugriffe auf die Hardware zu reduzieren. Dabei wird einerseits eine vereinfachte Repräsentation der Szenengraphknoten im Bildraum und andererseits die Informationen von bereits durchgeführten Verdeckungstests verwendet. Da alle vorgestellten Algorithmen in irgendeiner Form von den Tiefenwerten von bereits gezeichneten Teilen der Szene abhängen, ist die Reihenfolge der Darstellung und der Verdeckungstests von zentraler Bedeutung. Im dritten Teil der Arbeit wird daher ein neuer Traversierungsalgorithmus für den Szenengraphen vorgestellt, der die von der Graphikhardware zur Verfügung gestellten Verdeckungstests besonders effizient ausnutzen kann. Dazu sucht der Algorithmus nach Kohärenzen im Bildraum in Kombination mit einer sortierten Traversierung der Knoten im Objektraum. Um letztendlich die Verdeckung eines Objekts zu bestimmen, bündelt der Algorithmus die Verdeckungstests in Mehrfachanfragen, die es erlauben, mehrere unabhängige Verdeckungstests asynchron durchzuführen.

Ganz bewusst wurde auf die Verwendung von speziellen – räumlichen – Datenstrukturen für die Szene verzichtet, um lange Vorberechnungszeiten oder Einschränkungen für dynamische Szenen zu vermeiden. Ebenso wurde darauf verzichtet, zeitliche Kohärenz zwischen einzelnen Bildfolgen auszunutzen, da dies Einschränkungen für interaktive Szenen zur Folge hätte. Gleichwohl erlauben die vorgestellten Algorithmen eine effiziente Darstellung beliebiger Szenen mit hoher Tiefenkomplexität.

Contents

1 Introduction **3**

1.1 Motivation 3

1.2 Contribution of This Thesis 4

1.3 Structure of the Thesis 5

2 Background **7**

2.1 Scene Organization 7

2.1.1 Space Subdivision 8

2.1.2 Scene Graphs 9

2.1.3 Bounding Volumes 14

2.2 Rendering 15

2.2.1 Rasterization Pipeline 15

2.2.2 Multi-Resolution 16

2.2.3 Visibility Culling 17

3 Occlusion Culling **21**

3.1 Introduction 21

3.2 Classification 24

3.3 Hardware Occlusion Queries 26

3.4 Related Work 28

4 Occlusion Query Implementations **33**

4.1 Introduction 33

4.2 Using the OpenGL Depth Buffer 34

4.3 Using the OpenGL Stencil Buffer 35

4.4 OpenGL Extensions for Occlusion Culling 36

4.5 Evaluation 37

5 Avoiding Predictable Occlusion Queries **41**
5.1 Introduction 41
5.2 Occupancy Map 42
5.2.1 Implementation 42
5.3 Additional Depth Buffer in Software 44
5.4 Temporal Coherence 45
5.5 Results 46
5.5.1 Occupancy Map and Software Depth Buffer 50

6 Occlusion Driven Traversal **53**
6.1 Introduction 53
6.2 Occlusion Query 54
6.3 Organization of Multiple Occlusion Queries 55
6.4 Traversal 56
6.4.1 Complexity 62
6.5 Results 63
6.5.1 Scene Graph Structure 66

7 Conclusion and Future Work **69**
7.1 Conclusions 69
7.2 Main Results 70
7.3 Future work 71

8 Appendix **73**
8.1 OSGViewer 73
8.1.1 Implementation 74
8.1.2 Main Part 74
8.1.3 Scene View 75
8.2 Test Models 77
8.2.1 Cotton Picker 77
8.2.2 Formula One Car 78
8.2.3 City 80
8.2.4 Boom Box 81

Bibliography **83**

List of Figures

1.1 A complex MCAD model of a Cotton Picker. 3

2.1 Structure of a 3d scene subdivided in its objects. 7
2.2 Examples of an octree and BSP tree 8
2.3 Example of a simple scene graph. 9
2.4 Internal data structure of a Jupiter scene graph. 11
2.5 Internal data structure of an OpenSG scene graph. 13
2.6 Bounding volume hierarchy for the Formula One Car. 14
2.7 Block diagram of the OpenGL rendering pipeline. 15
2.8 Two different resolutions of the same model. 16
2.9 View frustum culling and backface culling. 18
2.10 Contribution or detail culling. 19

3.1 Visible and occluded objects of a scene. 21
3.2 Occlusion culling. 22
3.3 Depth complexity. 23
3.4 Point- vs. region-based occlusion culling. 24
3.5 Block diagram of the OpenGL occlusion query extension. 26
3.6 Pipeline of rendering with occlusion queries. 27
3.7 Hierarchical z-buffer. 28
3.8 Hierarchical Occlusion Maps 29
3.9 Illustration of the depth estimation buffer. 30
3.10 Working set used for temporal coherence. 31
3.11 Differences between a kd-tree and a scene graph. 32

4.1 Overview of the software architecture 33
4.2 Block diagram of the OpenGL z-Buffer read caching scheme. 34
4.3 Z-buffer with marked fragments. 35
4.4 Occlusion test with the stencil buffer. 36
4.5 Block diagram of the OpenGL stencil buffer read. 36

4.6 Latency of the HP Occlusion Flag 37
4.7 Rendering results. 38

5.1 Overview of the software architecture 41
5.2 Construction of the Occupancy Map. 42
5.3 Example for an Occupancy Map. 43
5.4 Request to the Occupancy Map. 44
5.5 Occlusion test with the Software Depth Buffer. 45
5.6 Experimental results. 47
5.7 Experimental results. 48
5.8 Experimental results. 49
5.9 Experimental results. 51
5.10 Experimental results. 51
5.11 Experimental results. 51

6.1 Overview of the software architecture 53
6.2 Multiple query without and with possibly redundant results. 54
6.3 Request to the Occupancy Map. 55
6.4 Traversal of the scene graph. 57
6.5 Traversal – Stage 1. 58
6.6 Traversal – Stage 2. 58
6.7 Traversal – Stage 3. 60
6.8 Occluded bounding boxes of multiple occlusion queries. 61
6.9 Worst case situation for the traversal 62
6.10 Boom Box frame rates and rendered polygons. 64
6.11 Frame rates and rendered polygons for the different models. 65
6.12 Rendering performance depending on scene graph depth 67

8.1 Screen shot of the OSGViewer. 73
8.2 Main window of the OSGViewer. 74
8.3 Scene view of the OSGViewer. 75
8.4 Screen shot of the statistics window. 76
8.5 Complex scene of a Cotton Picker. 77
8.6 Depth complexity of the Cotton Picker model. 77
8.7 Formula One Car. 78
8.8 Depth complexity of the Formula One Car model. 78
8.9 Screen shots of the F1 Animation model. 79
8.10 Some frames of the F1 Animation. 79
8.11 Different views of the Big City model. 80

8.12 Different views of the Boom Box model. 81

Acknowledgments

First, I would like to thank my advisor, Prof. Wolfgang Straßer. This work would not have been possible without his long term support. I would also like to thank Prof. Andreas Schilling for his additional support of my work. My thanks go to Dirk Bartz and Michael Meißner for their help on writing papers and the discussions about research stuff. Special thanks go to my colleagues for their support in a lot of different situations, especially Michael Wand for the interesting discussions regarding rendering, operating systems and programming languages.

The work in this thesis is based on the OpenSG scene graph, which was initially introduced by Dirk Reiners, Gerrit Voss and Johannes Behr. The OpenSG PLUS project, which the work in this thesis is part of, was funded by bmb+f based on OpenSG.

English is a foreign language for me and I wish to thank Maria Finnegan for her corrections and help on language issues.

Finally, thank you to my family and my girlfriend Melisa Mekic for their patience and aid.

Chapter 1

Introduction

The first Chapter explains the motivation of the presented work, provides an overview of the contributions and a brief outline of the structure of the thesis.

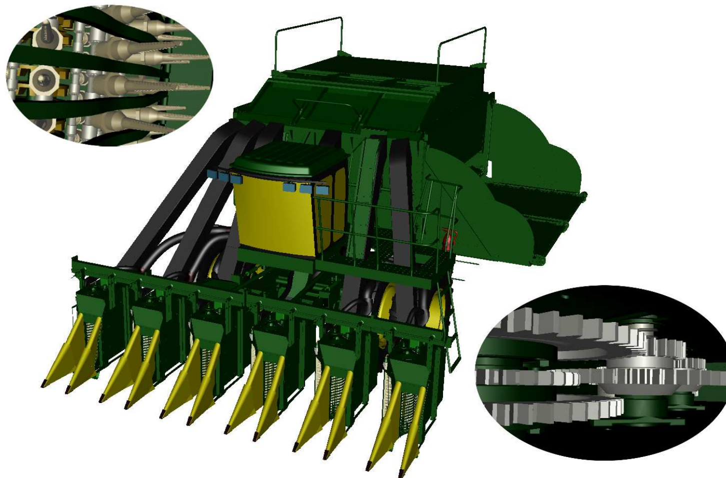

Figure 1.1: A complex MCAD model of a Cotton Picker. The model has a complexity of over 10 million polygons in 13 270 individual parts.

1.1 Motivation

The datasets for visualization are growing faster in size than the rendering speed of modern graphics subsystems. Especially due to the increasing capabilities of 3d scanner hardware [4], very complex data sets have become available. But also artificial/constructed data sets from MCAD (see Figure 1.1) or from physical simulations

have become more complex due to increasing computational power. For example, the rendering of a Boeing 777 would include 132,500 unique parts and over 500,000,000 polygons [21]. Acceleration algorithms will always be necessary to handle such large amounts of data.

Several algorithms already exist to address the problem of rendering such large data sets. Most of them reduce the number of primitives, others use sampling techniques such as raytracing [52] or point sampling [56, 57]. To reduce the number of primitives, level-of-detail [26] or impostor [22] techniques can be used. Another approach is occlusion culling, which is the focus of this thesis. As part of this process, occluded parts of a complex scene are detected and excluded from the rendering process. In particular, the rendering of scenes with high depth complexity can benefit from occlusion culling.

In contrast to many other occlusion culling techniques, *arbitrary* and *dynamic* scenes are also the focus of this work with regard to applying the techniques described here to a standard scene graph. A standard scene graph must be usable in a wide range of different applications, models, and platforms. To achieve this goal, miscellaneous capabilities of the graphics hardware should be usable to support a wide range of different hardware platforms. Also, precalculations or the construction of special data structures should be avoided to support "real-time" use. This is important for CAD systems or other interactive systems, where the user is able to modify the scene in real-time.

1.2 Contribution of This Thesis

The work in this thesis focuses on occlusion culling for common scene graph systems, which has the previously described implications: support for different hardware platforms, no time for preprocessing and support for general and dynamic scenes. The thesis is subdivided into three main parts, which complement to each other:

- We present and compare different techniques for the exploitation of standard z-buffer hardware for occlusion culling [6]. This includes the presentation of a new caching scheme for z-buffer accesses and a quantitative analysis of hardware supported occlusion queries.

- Furthermore, we also present some algorithms to reduce the number of predictable occlusion queries to the graphics hardware [5, 8]. Object space coherence and sorted rendering is used to achieve this goal. The reduction of the

number of occlusion queries is important to decrease the needed latency for the hardware communication.

- The main contribution is a new traversal and sorting algorithm that benefits from *multiple* occlusion queries of modern graphics hardware, which is available on a wide range of graphics hardware and part of the OpenGL 2.0 standard [9, 10, 12]. The algorithm significantly reduces the latency overhead of occlusion queries. Also no preprocessing or spatial data structures are necessary, so that the algorithm can be directly used with a scene graph system to benefit from hardware occlusion queries.

1.3 Structure of the Thesis

This thesis is organized as follows; the next Chapter describes the background for this thesis and gives a short introduction to scene graphs and acceleration techniques for rendering. A more detailed description of occlusion culling and related work to this thesis is then given in Chapter 3. Chapter 4, 5, and 6 present the practical part and corresponding results of this thesis. Different implementations of algorithms to determine the visibility are described in Chapter 4, followed by extensions to prevent occlusion queries in Chapter 5, and concluding with a special traversal algorithm in Chapter 6. The thesis closes in the final Chapter with a summary of the presented work and an outlook on future work. An overview of the application and test models used for the experiments can be found in the Appendix.

Chapter 2

Background

This Chapter gives a short introduction to the necessary background for the work in this thesis, which includes data structures for scene organization and different acceleration methods for rendering, such as multi-resolution and culling. The main focus is on scene graphs, real-time rendering and visibility culling techniques.

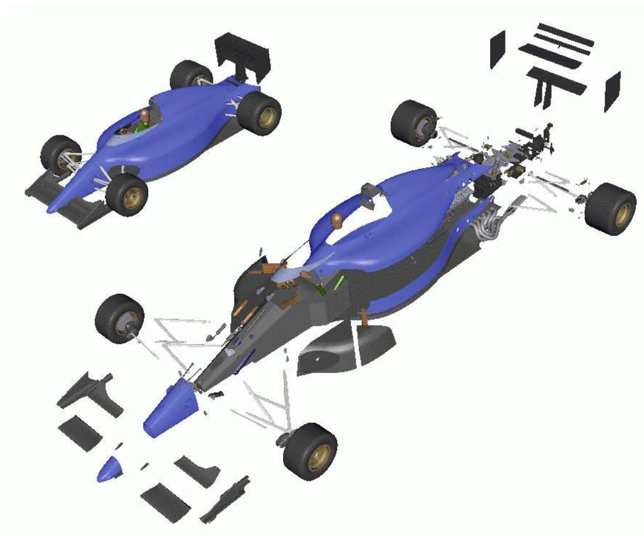

Figure 2.1: Structure of a 3d scene subdivided in its objects.

2.1 Scene Organization

Usually a 3d graphics scene consists of a large number of polygons (triangles), however other primitives like points, voxels or splines are also used. The primitives in

3d space are defined by vertices, normals, texture coordinates, etc. To organize and manage this data, hierarchical data structures are used [24, 35, 36]. Hierarchical data structures are able to reduce the complexity from $O(n)$ to sub-linear for a wide range of algorithms, like picking or view frustum culling. The data structures can be classified into two different types, with and without space subdivision [48]. Examples for hierarchical data structures with space subdivision are octrees or BSP trees. In contrast, scene graphs are data structures without strict space subdivision.

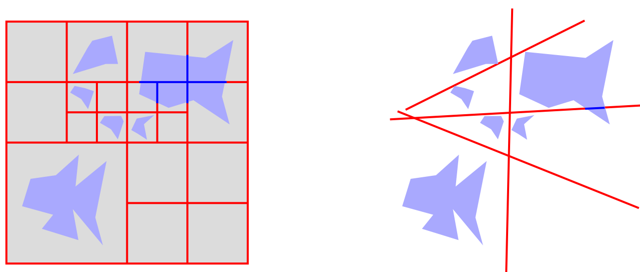

Figure 2.2: 2D examples of an octree (left, the term *quadtree* is used in 2D) and a BSP tree (right).

2.1.1 Space Subdivision

Octree

An octree [36] recursively subdivides the 3d space into eight, equal sized boxes (see left Figure 2.2). The 3d informations are sorted into these boxes. The advantage of an octree is its spatial representation, which is useful for multi-resolution rendering, spatial coherent traversal and data compression [13, 28]. The drawback of an octree is the cost for an update in dynamic scenes and the strict subdivision of space.

Binary Space Partitioning

A binary space partitioning (BSP) tree [24] recursively subdivides the 3d space with planes (see right Figure 2.2), which results in a binary tree representation of the scene. A BSP tree is useful for sorted rendering and visibility calculations. There are two variants, *axis-aligned* (*kd-tree*) and *polygon-aligned*. The construction of an effective BSP tree is a very time-consuming process, because it is hard to find "good" planes for subdivision. Thus BSP trees are usually precalculated once and stored for reuse.

Conclusion

Both the octree and BSP tree are useful to speed up rendering. However, the drawback here is that updates of the hierarchy become expensive, which makes handling of dynamic scenes difficult. Usually these data structures are calculated in a preprocessing step. Also space subdivision data structures are optimized only for geometry, other aspects like material sorting are not taken into account.

2.1.2 Scene Graphs

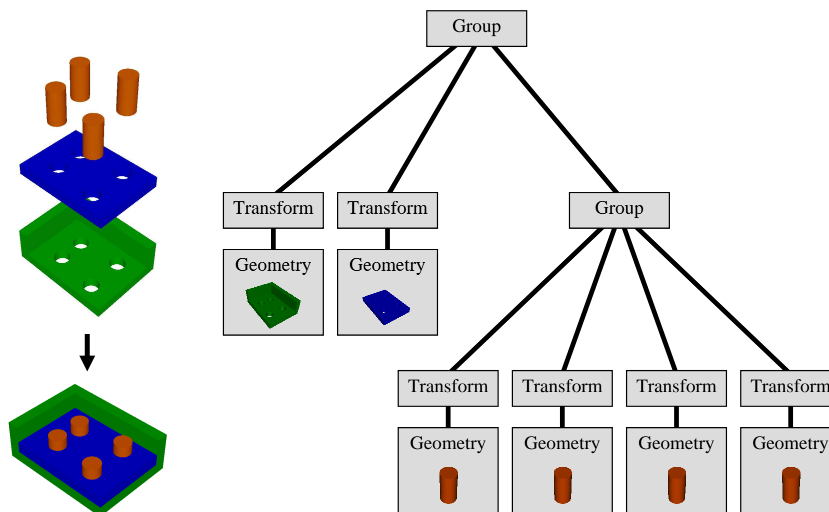

Figure 2.3: Example of a simple scene graph of the left model.

Scene graphs are used to represent a 3d scene in a hierarchical tree structure (see Figure 2.3). The tree structure has no space subdivision in contrast to octrees or BSP trees, which makes scene graphs handy for data manipulation, storing and organization (usually the scene is separated into objects like screws, nuts and springs, which is very useful in many applications like MCAD assembly or modeling tools, see Figure 2.1). The drawback is that there is no optimization for rendering in the data structure. However, a hierarchical structure of bounding volumes are usually supported by most scene graphs. Also, a scene graph is more like a database, which represents a scene. It is easy to add additional information to a scene, for example, identifiers or material parameters.

A scene graph consists of different types of *nodes*, for example, group, transform or geometry nodes. A set of 3d primitives construct a geometry node. The position in

the scene of a geometry node is driven by transform and group nodes, which arrange a set of child nodes. There are different types of scene graphs. Multi-parent scene graphs allow multiple parents for a node in contrast to single parent graphs. Multi-parent graphs allow a more flexible instantiation of the child nodes, whereas single parent graphs are easier to traverse and manage. Most scene graphs combine both techniques and scene graphs in combination with space subdivision data structures are also available [56].

Three main steps are performed to render an image of a scene graph. *Traversal*, *culling* and *rendering*. Usually these steps are interleaved to balance the load between the rendering subsystem and the host system, which performs traversal and some parts of the culling. In a simple implementation of a scene graph renderer, a recursive, stack-based, depth-first traversal of the graph is used. During traversal, material and transformation informations in the nodes are accumulated and culling is applied. If a node contains geometry, it is directly sent to the graphics subsystem and rendered.

Scene graph programming libraries are widely available and have a long tradition. Well known are Open Inventor [59], IRIS Performer [46], Cosmo3d [34], *Jupiter* [32, 3] or *OpenSG* [25], but there are many others. Almost every graphics application employs a scene graph in some way.

The main differences are the underlying data structures, support for large data sets, the software architecture, the used programming language and the flexibility of the traversal techniques. All of these scene graphs use polygons as main primitives and OpenGL [61] or Direct3d [53] for rendering, which does not mean that they cannot be used for raytracing or point sampling. Most of the scene graph toolkits are implemented in C++ with the advantage of performance and object orientation.

This thesis focuses on the scene graphs *Jupiter* and *OpenSG*, but all of the algorithms presented could also be used with other scene graph systems.

Jupiter

Jupiter [3, 32] focuses on large model rendering and provides different concepts for managing large amounts of data. Jupiter is based on a software initiative of Hewlett Packard (HP) and Engineering Animation Inc. (EAI) which led early in 1997 to the large model toolkit Jupiter, formerly also known as "DirectModel" [21] (the rights to the name "DirectModel" were later acquired by Microsoft). This initiative was canceled later the same year, in favor of the "Fahrenheit" project by HP, Microsoft and SGI. EAI continued working on Jupiter and with the virtual halt of the Fahrenheit project in 1999, Jupiter was relaunched by HP as an Open Source project (Kelvin [62]) in conjunction with the department for Graphical-Interactive Systems

at the Wilhelm Schickard Institute for Computer Science (WSI/GRIS) at the University of Tübingen.

Originally, Jupiter was developed as platform and graphics application programming interface (API) independent toolkit, available with support for StarBase [30], OpenGL [61] and Direct3d [53] on Microsoft Windows and UNIX systems. In the version available from WSI/GRIS, Jupiter focuses on OpenGL and UNIX systems (Linux, Irix, HP/UX).

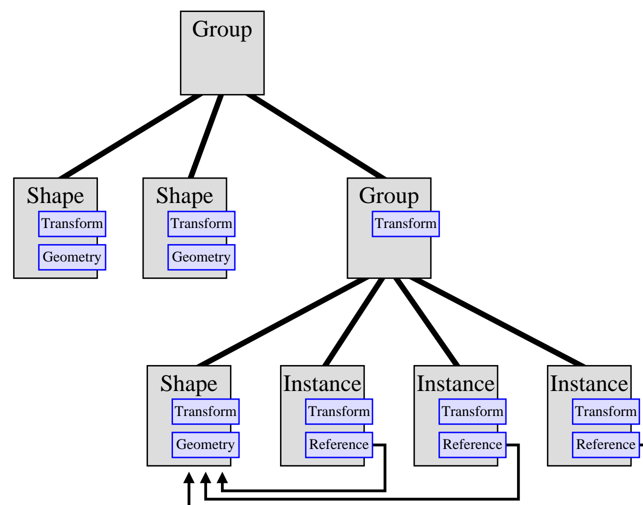

Figure 2.4: Internal data structure of a Jupiter scene graph.

The scene graph of Jupiter is called *logical graph* and is an acyclic directed graph. It consists of a variety of different nodes, which divide the partition of the model into objects and groups of objects (see Figure 2.4). Each node can contain a transformation matrix to specify a 3d location. The Jupiter scene graph is a single parent graph, but special instance nodes allow the same instance of a geometry node to be used multiple times in the graph to save memory. Geometry nodes are called *shape* nodes in Jupiter and contain different types of representations such as triangle strips, polygon sets or polyline sets. To manage large models, *level-of-detail* nodes handle subgraphs of different resolutions of geometry. A special *partition* node is also available, which specifies out-of-core subtrees. These subtrees can be swapped from or to the disk if necessary.

Jupiter has a very flexible concept for the traversal of the scene graph. A special class, called *strategy*, is used to manage the budget-based traversal. Each strategy consists of a set of pipelined *agents* which evaluate the importance of the scene graph nodes and manipulate the traversal accordingly. The importance can be based on the

visibility of a node, on its visual contribution, or on specific properties of a node. The agents process each node in a pipelined fashion. The order of the pipeline is important and depends on the performance of an agent. For example, the view frustum culling agent is used in front of the rendering agent. Jupiter provides agents for several purposes, like memory management, picking, rendering, culling.

A strategy can be used for different tasks, like rendering or picking. The order during traversal is determined by a priority for each node. This allows different schemes for traversal, for example, object-space front-to-back, material sorted or image-space sorted. A more detailed description of Jupiter can be found in [3, 9, 32, 62].

OpenSG

While Jupiter has a longer tradition and started as a commercial product, OpenSG started in 1999 as Open Source project in Darmstadt, Germany, after the cancellation of Fahrenheit [25]. In conjunction with the OpenSG project, the OpenSG PLUS project started 2001 with support by the bmb+f[1]. OpenSG PLUS adds support for large scenes, high level primitives and high level shading [20] to OpenSG, which provides core functionality.

The focus of OpenSG is on a rendering system for virtual reality (VR) applications. There are a wide range of VR applications. Often special hardware configurations are used to give better impressions in the virtual environment. This means that high-performance graphics hardware is needed. To obtain maximum rendering performance and quality, a cluster can be used for rendering, especially if multiple projectors for stereo, CAVEs or high resolution displays are to be operated. Each projector is driven by its own computer and the computers are connected to a cluster. OpenSG can manage and distribute the rendering over such a cluster. To achieve this goal, OpenSG uses a more complex scene graph model with reflection and thread-safe data structures.

Reflection means that each instance can provide information about its internal data structures during run-time of the application. Thread-safe means that asynchronous modifications of the scene graph data are tracked and synchronized in a cluster or multi-threaded environment.

In contrast to Jupiter and many other scene graphs, the graph and geometric, material, transformation, etc. information is separated in OpenSG. Each scene graph node is the same node type and has a pointer to its parent, its children and its *core*. The *core* contains the actual data, like transform matrices, material informations or

[1] Federal Ministry of Education and Research, Germany

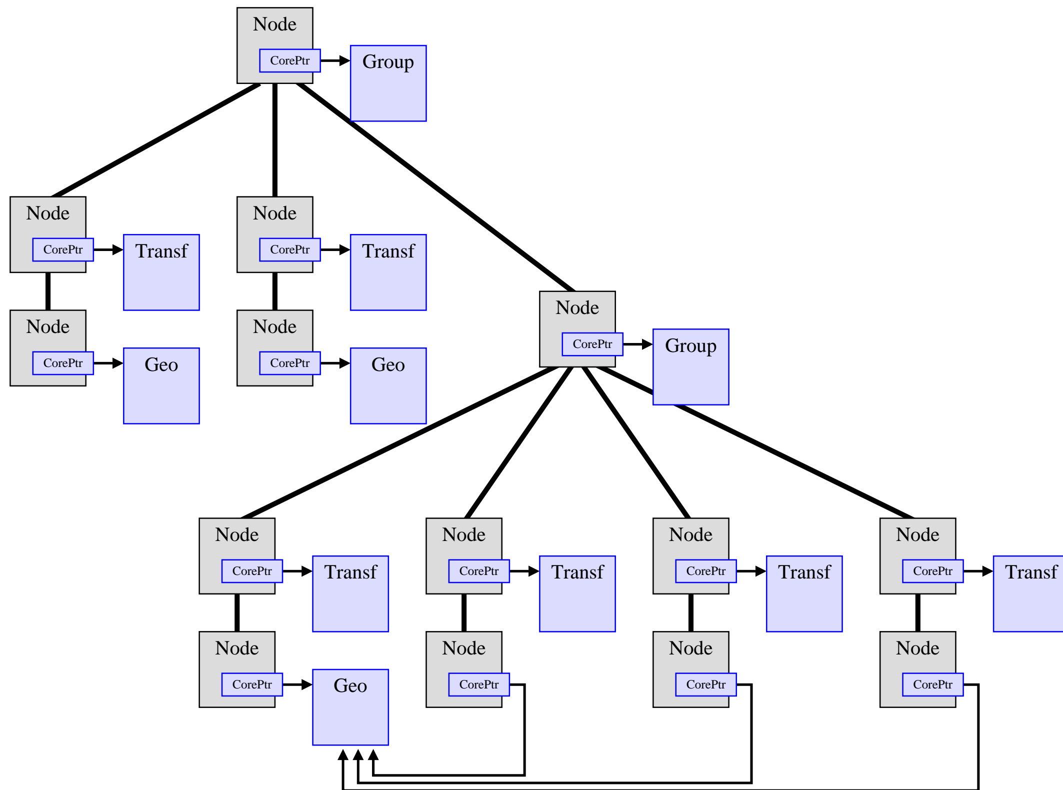

Figure 2.5: Internal data structure of an OpenSG scene graph.

geometric data (see Figure 2.5). The OpenSG scene graph is also an acyclic directed graph with single parent nodes. But a *core* can be referenced by multiple nodes, which results in instantiation to save memory consumption. The background for this design decision is that the scene graph itself is very small in contrast to the geometric data. So the whole scene graph can be distributed and very easily updated over a cluster. On the other hand, the amount of geometric data can become very big and it is not necessary to distribute and update all data on every cluster node. At a first view, the OpenSG scene graph looks much more complex than the Jupiter or other scene graphs, but the internal data like matrices, geometry, materials, etc. are also stored as references in other scene graphs. OpenSG abstracts this structure to the application programmer to benefit for a better synchronization in cluster and multi-threading environments.

OpenSG focuses on OpenGL as rendering backend and polygons as main primitives. But other primitives like points or NURBS [37] are also supported, even volume rendering is possible [58]. Shader programming is also supported and can be

done by Cg [23] or GLSL [47]. Different operating systems like Linux, Windows or Irix can be used as platform for OpenSG. A more detailed description of OpenSG can be found in [45, 55].

2.1.3 Bounding Volumes

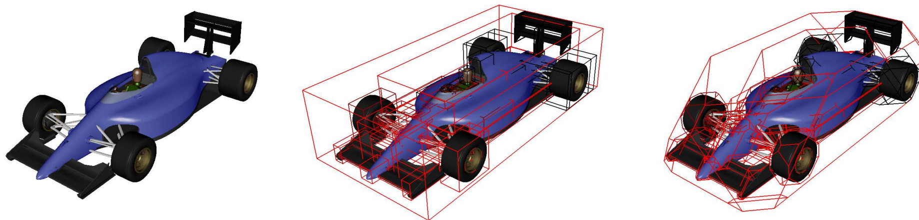

Figure 2.6: Bounding volume hierarchy for the Formula One Car (left: none, middle: bounding boxes, right: k-dops).

A *bounding volume* [35, 24] wraps a set of 3d primitives. Bounding volumes are very important to speed up a wide range of computations in computer graphics. Especially in collision detection and raytracing, but also in visibility and multi-resolution algorithms, bounding volumes are used to get a fast information about the spatial organization of an object. Often bounding volumes are grouped together to form a bounding volume hierarchy. Different types of bounding volumes are used, like spheres, boxes or k-dops (see Figure 2.6). Also they can be distinguished between a representation in local or global coordinates and axis- or local-aligned.

Usually every scene graph provides a bounding volume hierarchy. OpenSG and Jupiter provide bounding spheres and bounding boxes in each node. One specialty is the used coordinate system for the vertices. Only axis-aligned bounding volumes in local and global coordinates are provided, because they can be calculated in $O(n)$. In global coordinates each vertex of the underlying has to be transformed to global coordinate space. Local-oriented bounding volumes are much more complicated to compute, so that an update of a bounding volume comes more expensive.

In the latter, k-dops become more popular in computer graphics for a wide range of applications. They combine a fast calculation with a tighter approximation of the underlying object and it is not complicated to build a k-dop hierarchy. K-dops also becoming more popular in occlusion culling approaches [1, 2, 16].

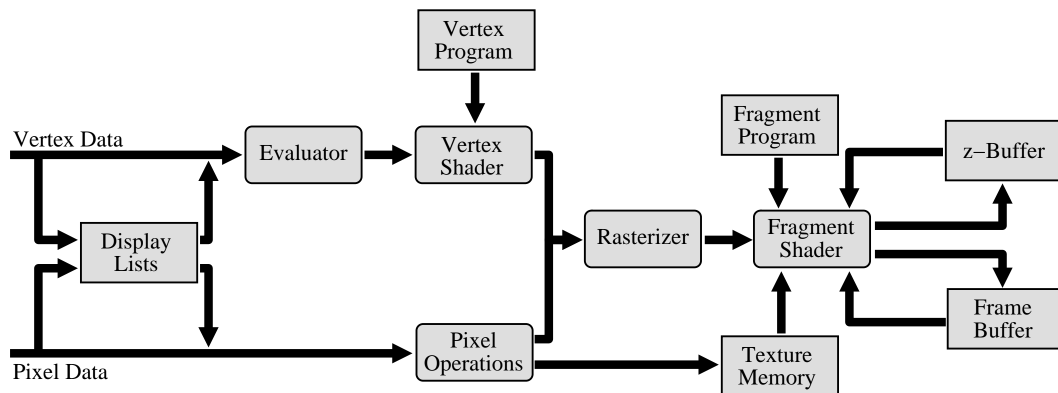

Figure 2.7: Block diagram of the OpenGL rendering pipeline.

2.2 Rendering

There are a wide range of rendering techniques [24, 35, 36] in computer graphics. Most popular are raytracing and polygon rasterization. Volume rendering, point rendering and image based rendering are other well known techniques. While raytracing is usually used for high quality rendering, polygon rasterization is well supported by special graphics hardware for real-time rendering.

2.2.1 Rasterization Pipeline

Usually this simplified (OpenGL) pipeline is used for polygon rasterization:

Classical Graphics Pipeline	*Programmable Graphics Hardware*
Triangle setup	Setup
Vertex transform	Vertex Shader
Lightning	Rasterization
Rasterization	Fragment Shader
Frame and z-buffer updates	Frame and z-buffer updates

The work of this thesis based on such a pipeline and reduces the load during rendering. Figure 2.7 shows a more detailed block diagram of the OpenGL pipeline. There are different types of bottlenecks in the pipeline:

1. Memory bandwidth host ↔ graphics board: transfer of the vertex data (including colors, texture, vertex coordinates, etc.).

2. Transformation and vertex shader performance.

3. Rasterization and fragment shader performance.

4. Memory bandwidth graphics chip $\leftrightarrow$ frame, texture and z-buffer.

5. State changes: changing configurations of the rendering like blending, lightning or different shader programs can cause graphics pipeline flushes and stalls. In addition to the latency caused by the graphics hardware, this also includes latencies from the operating system (interrupt handling, etc.) and graphics driver.

A lot of different acceleration techniques (in hard- and software) exist to reduce these bottlenecks. Most of them can be classified in two orthogonal types: culling and multi-resolution techniques.

2.2.2 Multi-Resolution

Multi-resolution or Level-of-Detail (LOD) algorithms try to reduce the number of used polygons and textures for rendering without loosing image quality by using simpler representations of an object if its contribution to the image is small. For example, consider a very detailed Cotton Picker model shown in Figure 2.8 that consists of 11,000,000 polygons in full resolution. The full resolution can be used, if the viewer is close to the model. But if the viewer is far away and the model covers only 20×20 pixels on the screen, 11,000,000 polygons are obviously not necessary to render a nice image (see Figure 2.8). Multi-resolution techniques are very useful in scenes with a lot of small, visible details, for example, trees in a forest or people in a stadium.

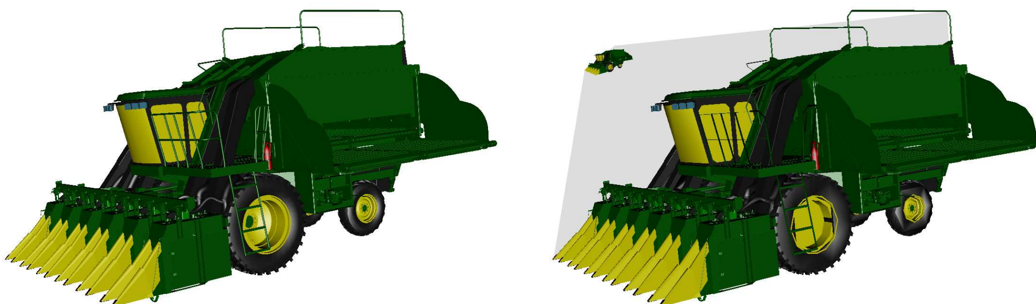

Figure 2.8: Two different resolutions of the same Cotton Picker model, left: full resolution with over 11 million polygons right: lower resolution with almost 3 million polygons.

Multi-resolution algorithms can be split into three major parts: *generation*, *selection* and *switching*. Generation is the part, which calculates different detail levels of an object. This can be done in a preprocessing step or during run-time. Selection decides, which detail level is used during rendering. Often, the estimated area in screen-space is used as criteria for selection, but other criterias like the distance to the viewer or a time-budget for rendering can be used. Finally, the change between different detail levels is termed switching. A switch between two detail levels is often noticeable, so different techniques try to reduce this effect.

Multi-resolution techniques are not the focus of this thesis, a more detailed description can be found in [26]. However, some visibility techniques can be used to give a hint in detail level selection. It is possible, to acquire the amount of visible pixels in relation to occluded pixels, which is useful for selection of an adequate object resolution. Multi-resolution techniques reduce the load of the vertex transform/shader unit of the graphics hardware and the transfer of vertex data to the graphics hardware. Also if multi-resolution is applied to textures, less texture memory and bandwidth is needed.

2.2.3 Visibility Culling

Visibility Culling tries to find parts of the scene, which are not (or almost not) visible from a given viewpoint. Culling techniques are very useful in scenes with high depth complexity or vast environments, like city or architectural environments. Another application are MCAD models with very detailed interior, like machines or cars.

Culling saves rasterization bandwidth, transformation calculations and texture or geometry transfers to the graphics hardware. Also state changes and shader calculations can be reduced. To maximize speedups during rendering, usually multi-resolution and culling techniques are combined.

View Frustum Culling

With view frustum culling, parts of the scene outside the view frustum are removed from the rendering process. Usually only bounding volumes are used to calculate the visibility. A bounding volume can be in three different states; *inside*, *outside*, and *intersection* with the view frustum. Objects whose bounding volumes are outside the view frustum, are not rendered, bounding volumes with intersection are further processed in a hierarchical data structure or rendered, if they contain geometry (see left Figure 2.9). If a bounding volume hierarchy is used, the term hierarchical view frustum culling is used.

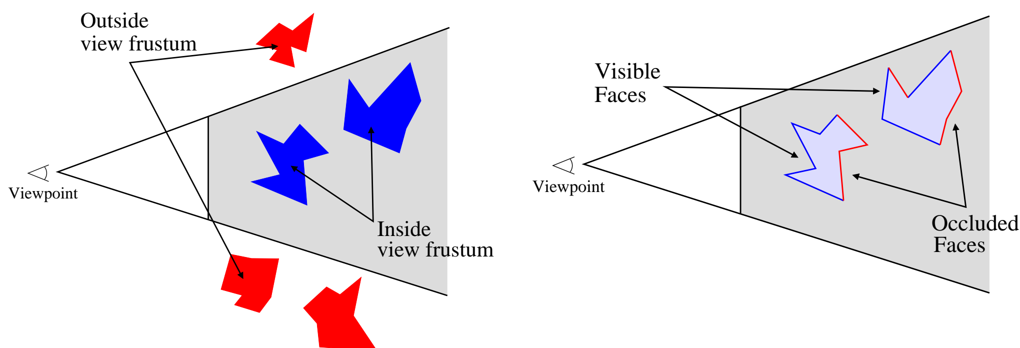

Figure 2.9: Left: view frustum culling, right: backface culling. The red parts are not rendered.

View frustum culling is commonly used and easy to implement, there are only a few minor difficulties. It is usually calculated on the host CPU during traversal of the scene data structure and thus, view frustum culling reduces the load of the graphics hardware.

Backface Culling

Opaque models of real-world objects are usually modeled with a back and a front side (see right Figure 2.9), but only the primitives which are facing to the viewer can be seen from a given viewpoint. Backface culling removes primitives, whose normals are pointing away from the viewer (backfaces) and thus are not visible (see Figure 2.9, on the right). This can be calculated by creating a vector d from an arbitrary point (for example, one vertex) on the polygon to the viewer: $d = v_0 - p_{viewer}$. Then compute the dot product of this vector with the normal of the polygon: $r = d \cdot n$. It follows that if $r < 0$ then $\sphericalangle(n,d) > \pi/2$, the polygon is not facing to the viewer. Another way is to use the normal n_p of the *projected* polygon; if $z_{n_p} < 0$ the polygon is not facing to the viewer, assuming that the negative z-axis is pointing into the scene.

Hardware solutions (supported by OpenGL backface culling) are using the orientation of the vertices of a polygon to estimate the projected normal of a primitive. This reduces the memory bandwidth on the graphics board since rasterization of backfacing polygons is not necessary, but the polygons are transfered to the graphics hardware and transformed. Also additional computations of the orientation may be necessary.

To save also bandwidth between the graphics board and the host, backface culling

can be done in software. But it is too expensive to test each normal of each primitive, hence clustering techniques can be used. A *normal cone* [51] is usually used, to cluster a set of primitives, whose normals are inside the cone. The cone is defined by a normal n and a half-angle α. The set of primitives is backfacing if the cone is backfacing.

Contribution and Occlusion Culling

Contribution (or Detail) Culling is making a trade-off between the contribution of an object to the resulting image and the image quality. If the contribution of an object to the resulting image is too small (see Figure 2.10), the object is not rendered. This is not conservative and results in (small) image errors, but is sometimes acceptable if the rendering speed is the important factor. Contribution Culling can be implemented by using the projected screen-space size of a bounding volume as degree of contribution. Often a major problem of contribution culling are not the image errors, but the flickering of appearing or disappearing geometry, which is similar to the switching of different level-of-details (cp. to Section 2.2.2).

Occlusion Culling is done, if an object has *no* contribution to the resulting image. This is the case, if an object is completely occluded by other geometry. This is a much more complex problem and thus a more detailed description about occlusion culling is given in the next Chapter.

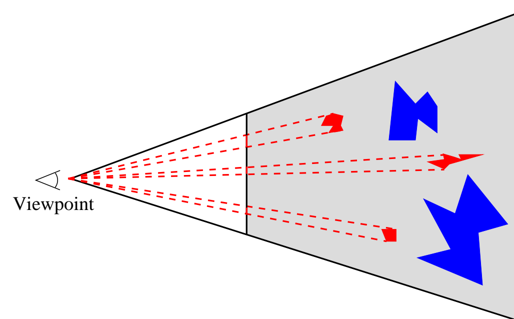

Figure 2.10: Contribution or detail culling, the red parts are not rendered.

Chapter 3

Occlusion Culling

Whereas the previous Chapter gives a more general introduction to the backgrounds of this thesis, the following Chapter focuses on occlusion culling in more detail. A classification of the different occlusion culling algorithms is given and related work to this thesis is presented.

Figure 3.1: Visible (left) and occluded (right) objects of a scene.

3.1 Introduction

Occlusion culling tries to find occluded geometry for a given viewport which is analogous to the visibility of an object. Occlusion culling is much more complex than the other visibility techniques, view frustum, backface and contribution culling. In contrast to the other culling techniques, occlusion culling is a *global* problem, due to the interaction of polygons of different objects, whereas backface, view frustum

or contribution culling can be calculated on a "per-object" basis, which is a *local* problem.

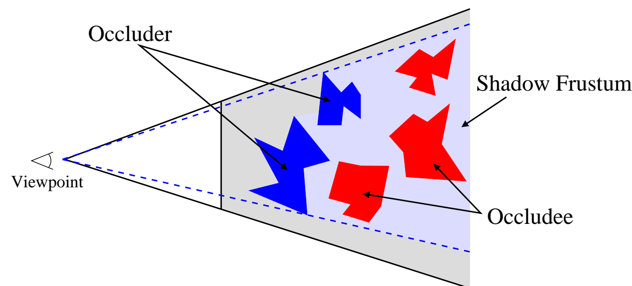

Figure 3.2: Occlusion culling. The red parts are not rendered.

Two major terms are commonly used in occlusion culling algorithms: *occluder* and *occludee*. *Occluders* are objects in the front of the scene, which occlude other objects. *Occludees* are objects behind occluders and not visible for a given viewpoint (see Figure 3.2). *Occludees* are in the *shadow frustum* of an occluder. If multiple occluder build a single shadow frustum is it called *occluder fusion*.

A *conservative* occlusion culling algorithm avoids only rendering of completely occluded objects. In contrast, an *aggressive* algorithm removes also possibly visible objects from rendering (cp. to Section 2.2.3).

Usually without occlusion culling, the hardware rendering backend calculates the visibility in a discrete way for each primitive with a z-buffer approach [17, 54]. Each primitive has to be scan converted to get the visibility of its pixels. Due to the linear complexity of the z-buffer, an efficient occlusion culling algorithm has to be sub-linear to gain a speedup. Especially scenes with high depth complexity are in the focus of occlusion culling (see Figure 3.3). In such scenes a lot of work of the z-buffer is wasted on occluded pixels from primitives that had to be transformed and scan converted.

Two different major approaches are used during rendering. Starting with a viewpoint v, an algorithm computes a set of visible objects O_v. These objects are then rendered with the graphics hardware:

```
// Listing 3.1

Viewport v;
VisibleObjects Ov;
Ov = getVisibleObjects(v);
```

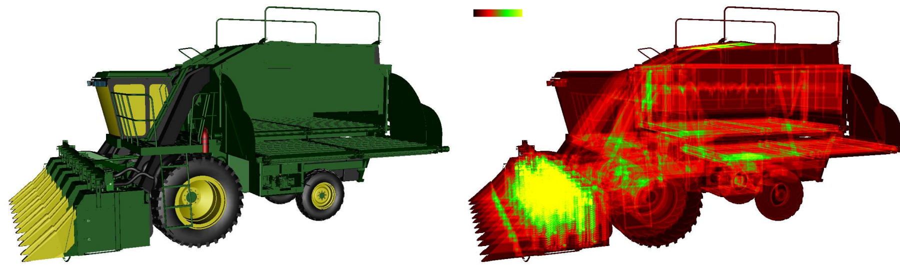

Figure 3.3: Depth complexity (averaged 13.4 z-buffer tests per pixel in 432 540 visible pixels).

```
for each object Ov[i] in Ov {
    Render(Ov[i]);
}
```

This approach is often used, if the visibility is precalculated in a preprocessing step or information of a previously rendered frame can be used. Another approach is to calculate the occlusion during rendering:

```
// Listing 3.2

Viewport v;
Objects O;

for each object Ov[i] in O {
    if(isOccluded(v, O[i]){
        Skip(O[i]);
    }else{
        Render(O[i]);
    }
}
```

The function `isOccluded()` is often called *occlusion test*, *visibility test* or *occlusion query*. Due to the complexity of visibility calculations, the test object itself is often not used to calculate its visibility in the occlusion test, but its bounding volume. In [1, 2] different types of bounding volumes are compared in the context of occlusion culling.

If a hierarchical data structure like a scene graph or octree is used, *hierarchical occlusion culling* can be applied. Each node of the graph can be used as entity for an occlusion query similar to hierarchical view frustum culling. The hierarchical

data structure can be subdivided into a visible and an occluded part. A subtree of an occluded entity is not further processed or traversed.

3.2 Classification

Occlusion Culling has a very long tradition in computer graphics, Cohen-Or et al. [19] give an overview and a classification of the occlusion culling techniques. They can be classified in point- vs. region-based, image- vs. object-precision and cell-and-portal vs. generic scenes. Additional criteria [19] to distinguish between algorithms are conservative vs. approximate, all vs. subset of occluders, convex vs. generic occluders, individual vs. fused occluders, 2D vs. 3d, special hardware requirements, need for precomputation and treatment of dynamic scenes.

Point- vs. Region-Based

Point-based algorithms are calculating the occlusion from a given, single viewpoint and view frustum. In contrast, region-based algorithms calculating visibility information for a whole region (see Figure 3.4). The viewpoint can be placed on any place in this region. Region-based occlusion culling is more complicated, because the shadow frustum of an occluder is no longer infinite, which makes occluder fusion much more difficult. However, the result of a region-based algorithm can be used over multiple frames until the viewpoint is leaving the region.

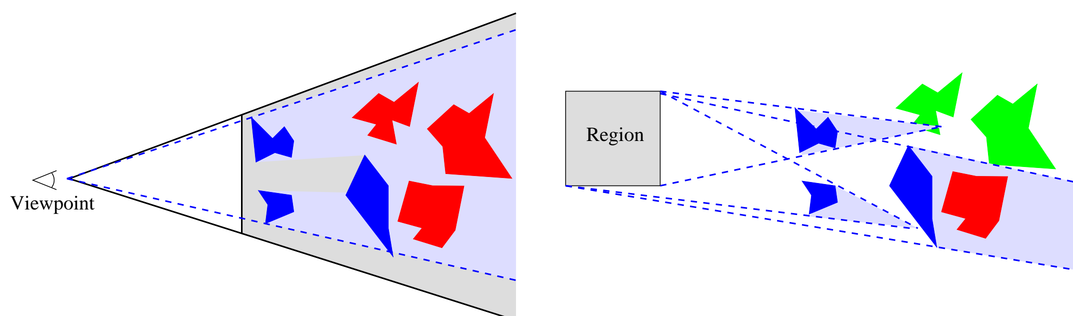

Figure 3.4: Point- vs. region-based occlusion culling.

Also point and region-based visibility can be compared with point and area light sources, where an occluded object is in the umbra (shadow frustum) region.

Object Space Precision

If an occlusion culling technique works in object space precision, each calculation is performed on the original 3d coordinates of the objects and returns a set of visible or occluded polygons. This is an expensive calculation and has a complexity of $\Omega(n^2)$ for a *given viewpoint* to get the *exact visibility set* (EVS). The EVS defines all primitives that are partially or fully visible.

The *aspect graph* is a data structure, which allows to reconstruct the EVS for every viewpoint in $O(1)$. Creating such a data structure is theoretically possible, but has a complexity of $O(n^9)$ [19]. For practical use, algorithms usually try to find a *Potential Visible Set* (PVS), which is a subset of the EVS and over-estimates the number of polygons, which are visible for a given viewpoint. The result of a PVS is usually rendered with a z-buffer algorithm to get a correct image [40].

Image Space Precision

Image-space precision visibility returns a discrete information of visible pixels (in place of polygons, which explains the lower complexity) for a given scene, camera and *resolution* of the viewport. Image-space algorithms are using the projected values of the original 3d coordinates.

The main advantage of image-space precision is the linear complexity with the z-buffer algorithm [29] and its hardware implementation. Almost every modern 3d graphics hardware uses the z-buffer to calculate the visibility of the primitives. A drawback of the z-buffer algorithm is the need to scan convert each primitive in full z-buffer resolution. In scenes with high depth complexity, a lot of pixels are rasterized and z-compared without a contribution to the resulting image. Also there is no feedback to the application thus every primitive has to be processed. This problem is addressed by modern graphics hardware, which implements a feedback to the application with the *hardware occlusion queries*.

3.3 Hardware Occlusion Queries

To speed up image-space occlusion queries, it is useful to use the graphics hardware in some way. Greene et al. [27] suggested a *feedback* for rendering of objects to get a faster occlusion information. Bartz et al. [14] proposed a detailed hardware implementation of occlusion queries. The first commonly available[1] graphics hardware with OpenGL extensions for occlusion queries was the VISUALIZE fx [49] from 1998 produced by Hewlett-Packard. A specification of the first OpenGL extension *HP Occlusion Flag* can be found in [31].

The fundamental idea is, to render a bounding volume through the graphics hardware with disabled frame and z-buffer writes. If a pixel triggers a z-buffer write during rasterization, a flag is set to true (see Figure 3.5). After rendering of the volume, the application can get the flag by the *HP Occlusion Flag* extension. If the result is true (at least one pixel of the bounding volume was visible), the content of the bounding volume has to be rendered or processed further. If the hardware occlusion query for an occluded bounding volume is cheaper than the rendering of the corresponding geometry, an application can be speeded up.

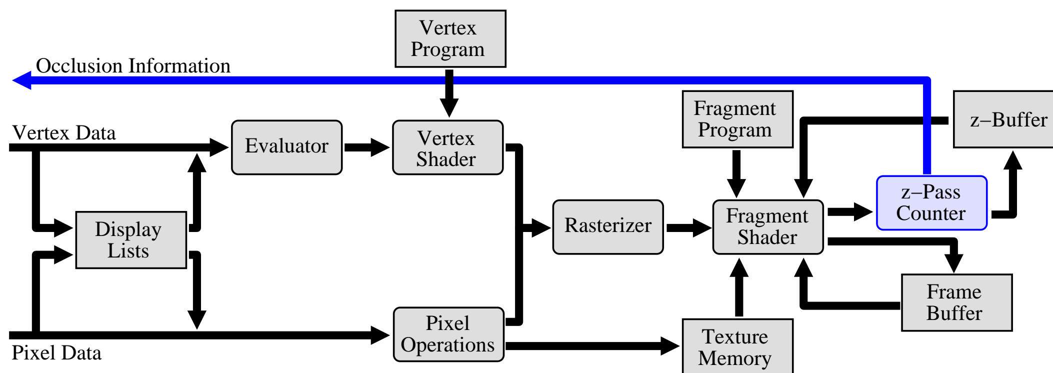

Figure 3.5: Block diagram of the OpenGL occlusion query extension.

The *HP Occlusion Flag* provides an easy way to get the occlusion information of a single geometry. The drawback here is that each request to the flag is synchronous. A new request can only be started after the finish of the previous one. This problem is addressed by the *HP Visibility Extension* [33] and by the more well known *NVidia Occlusion Query* [44]. Both extensions support *multiple occlusion queries*

[1]Maybe this was firstly implemented on a Kubota Pacific Titan 3000 with Denali GB graphics hardware [27] and should also be possible on a 3dfx voodoo add-on card from 1996, which supports pixel counting.

at the same time (see Figure 3.6). Additionally, the NVidia extension returns the amount of visible pixels of each tested geometry instead of a simple flag. This can be used for level-of-detail selection or contribution culling.

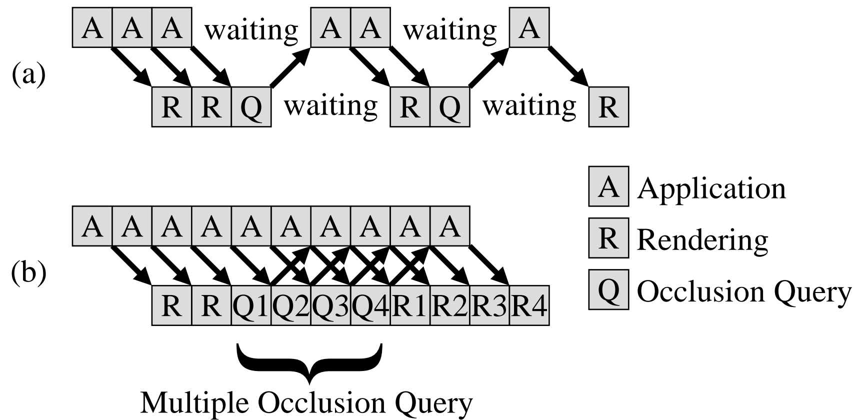

Figure 3.6: Pipeline of rendering with (a) synchronous single queries and (b) asynchronous multiple queries.

The vendor specific OpenGL extensions were added to the OpenGL 2.0 standard in 2004 [50]. Also a wide range of graphics hardware with support for the extensions are available, for example, NVidia GeForce3/4Ti or newer. OpenGL 2.0 or DirectX 9.0 capable systems like the ATI Radeon 9700 or the NVidia GeForce FX have to support the occlusion extensions to be fully compliant with the standards.

Discussion

The advantage of the occlusion queries using the current values of the z-buffer is a drawback at the same time, because the result of an occlusion query depends on up-to-date values in the z-buffer. The rendering system has to render an occluder A before the occlusion query Q_B for the occludee B is performed. If A is rendered *after* Q_B, the occlusion query will return visible as result for B, which is wrong and B would be rendered. Therefore the rendering sequence is important for the results of the occlusion queries. As expected, experiments [5] showed that a (partially) front-to-back sorted rendering gives good results.

3.4 Related Work

A lot of occlusion culling algorithms are available. Some of the occlusion culling techniques need extensive preprocessing [43, 40] or special scenes [41, 60]. These are not in the scope of this thesis. In the taxonomy of Cohen-Or et al. [18] this thesis focuses on conservative, from-point image-space approaches for generic scenes and generic occluders. The following algorithms are also image-based algorithms for real-time rendering. Further information on the various occlusion culling techniques can be found in [18, 19].

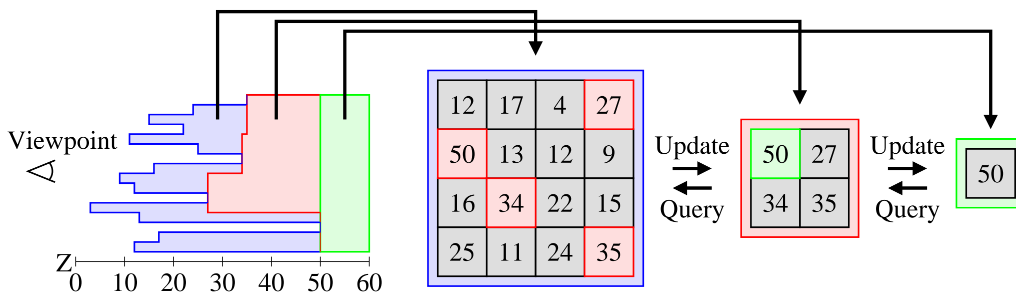

Figure 3.7: Hierarchical z-buffer.

One of the well known image-space algorithm for real-time rendering is the *Hierarchical z-Buffer* proposed 1993 by Greene et al. [27]. Many of the concepts of the algorithm had a significant influence on real-time occlusion culling research. The algorithm uses hierarchical data structures for the frame's z-buffer and the scene. The scene is organized in an octree and the depth buffer in a *z-pyramid* (see Figure 3.7). The finest level (highest resolution) of the z-pyramid is equivalent to the standard z-buffer. At the other levels, each z-value is the farthest z in the corresponding 2×2 environment of the next finer level. Whenever a z-value is overwritten in the z-buffer, it is propagated through the coarser levels of the z-pyramid. The hierarchical representation of the z-buffer (see Figure 3.7) reduces the number of z-tests during rasterization of a bounding box, which is used to perform the occlusion query for a node in the scene's octree. The hierarchy is maintained in software but Greene et al. suggested a hardware implementation for real-time rendering. To apply hierarchical occlusion culling, the scene's octree is traversed in a rough front-to-back order. The bounding box of an octree node is tested against the values in the z-pyramid to get its visibility. During z-rasterization of the box, the values are tested top-down against the z-pyramid. Only if the z-value would be visible in the corresponding z-pyramid level, the z-pyramid is traversed to the next level. If a z-value would also be visible in the finest level, the corresponding bounding box is visible. For visible octree boxes,

the testing continues recursively down in the octree. If a visible octree node contains geometry, the geometry is rendered into the z-pyramid to update the z-values and to use the geometry as occluders for subsequent occlusion queries. The main drawback of the Hierarchical z-Buffer is that an update to the frame's z-values results in an expensive update of the hierarchy, hence the Hierarchical z-Buffer is only useful for scenes with very high depth complexity. In such scenes the number of hierarchy updates are low in relation to the number of occlusion queries. The algorithm in Section 4.2 works similar to the Hierarchical z-Buffer but does not use a z-pyramid to avoid expensive hierarchy updates, because the time for an occlusion query with a visible result should be as low as possible to reduce the latency for the necessary subsequent rendering of the corresponding object. On the other hand, the time for an occlusion query for an occluded object has only to be lower than the rendering time for the corresponding object to gain a speedup (obviously each latency has to be low as possible to gain the best speedup).

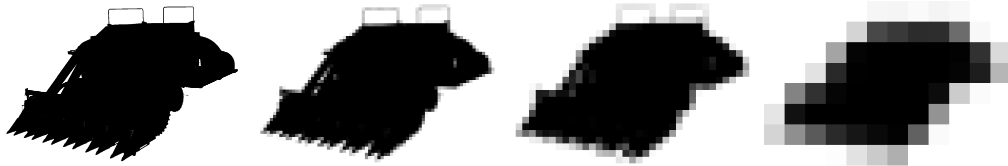

Figure 3.8: Hierarchical Occlusion Maps generated by the texture unit of the graphics hardware.

Another well known image-space algorithm are *Hierarchical Occlusion Maps* proposed 1997 by Zhang et al. [63]. They are using the graphics hardware to generate different levels of *occlusion maps* (see Figure 3.8). In contrast to the Hierarchical z-Buffer, Zhang et al. are using an overlap test in the xy-plane in combination with a subsequent depth test to get the visibility for an object. A set O of preselected occluders is used to generate the content of the first-level occlusion map. Then the texture unit of the graphics hardware is used to calculate the higher levels of the occlusion map hierarchy. A value in an occlusion map saves the average value of the underlying 2×2 pixels. In the highest resolution the occlusion map contains either white or black pixels describing the occupied regions of the preselected occluders. The greyscale of the values in the inner levels of the occlusion maps describe the "opacity" of the pixels. A high opacity value for a pixel in any level means that most of the pixels are occluded in the overlap test. Due to the missing depth information of the occlusion maps, a second data structure, the *depth estimation buffer* is used, which stores farest depth values of the occluders O in a lower resolution z-buffer (see

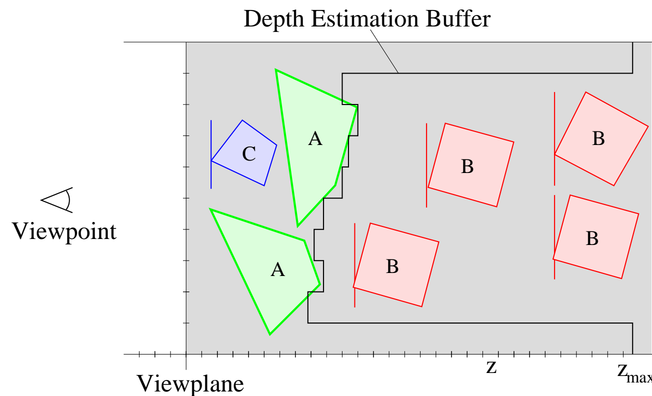

Figure 3.9: Illustration of the depth estimation buffer; A: Preselected occluders, B: Occluded objects behind the values of the depth estimation buffer, C: Visible objects in front of the depth values.

Figure 3.9). Before an object (usually a bounding volume) is tested with the depth estimation buffer, a two-dimensional, hierarchical xy-overlap-test is performed with the occlusion map (similar to the test with the hierarchical z-pyramid, but without z-values). Only if a bounding volume is occluded in the occlusion maps, a second test with the depth estimation buffer is performed. A main part of the algorithm is the selection of "good" occluders for generation of the occlusion maps and the depth estimation buffer. The algorithm benefits from the generation of the hierarchy by hardware and from a faster test against the occlusion maps (no z-values are needed in this part), but has the drawback that the occlusion efficiency depends on the quality of the preselected occluders. Also the lower resolution in the depth estimation buffer reduces the occlusion efficiency. The main difficulty of the algorithm is the selection of the occluder set, especially if preprocessing time should be avoided to support dynamic and modifiable scenes. This prevents the implementation in a commonly usable scene graph system.

Bartz et al. proposed a technique called *Virtual Occlusion Buffers* [15], which uses the OpenGL graphics pipeline to get image-space visibility information. A more detailed description can be found in Section 4.3.

Obviously, one of the fastest ways to utilize an occlusion query for geometry culling for general scenes is by special hardware support (see Section 3.3 and 4.4). But the use of this support is not for free, due to needed state changes and rasterization bandwidth on the graphics hardware. So there is a new field of research [19] to find ways for an efficient use of hardware occlusion query extensions, which is also the main part of this thesis.

Klosowski and Silva [38] have developed an algorithm, called *prioritized-layered projection algorithm* (PLP), which focuses on constant frame-rates using occlusion culling. However, their algorithm is not conservative, it sacrifices image quality in order to keep the constant frame-rate. Later, they have extended their algorithm with hardware occlusion queries to a conservative approach [39]. The algorithm has no restrictions for occluders and has occluder fusion, but needs time for preprocessing. In the preprocessing step the whole scene is partitioned into convex cells. The collection of the cells is generated in a way that each cell has a roughly uniform density of the primitives. This scheme results in larger cells in unpopulated areas and smaller cells in densely occupied regions. The original implementation used a Delauney triangulation, which was replaced by a more efficient octree implementation in newer versions. After the subdivision of the scene, a "solidity" value is generated for each cell, which represents the intrinsic occlusion. During rendering the traversal algorithm uses the solidity value in combination with the viewpoint and view direction to generate a prioritized list of cells which would be likely visible and therefore should be rendered. The original PLP algorithm is not conservative, because the rendering stops after a given budget. But the result is a good approximation of the image and therefore PLP can be used as occluder selection algorithm. To be conservative a subsequent test with hardware occlusion queries can be used to find the remaining, visible cells [39]. The main drawback of the algorithm is the preprocessing step which reduces the algorithm to static scenes.

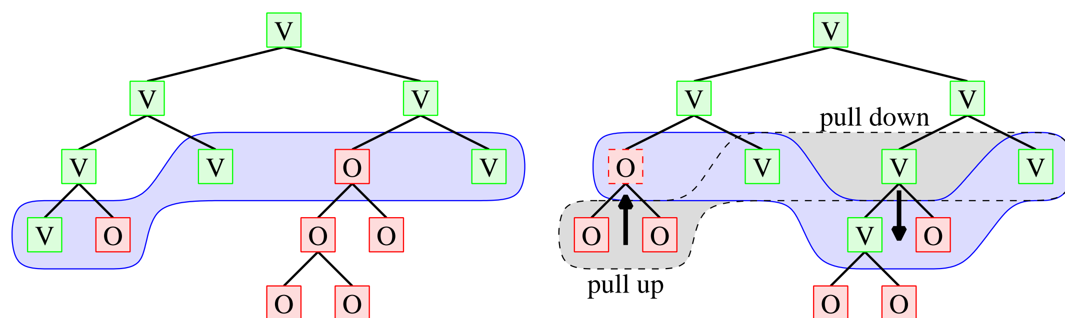

Figure 3.10: Working set (blue) which is used for an multiple occlusion query (left) and how the working set is changed between the frames (right) [16]. The nodes of the kd-tree are classified into visible (green) and occluded (red).

Bittner et al. [16] presented an algorithm, which uses particularly *temporal (frame-to-frame) coherence* to select bounding volumes for hardware occlusion queries. The scene is organized in a kd-tree, they call the used working set *termination nodes*. For each frame, a multiple occlusion query is performed for these termination nodes (see

Figure 3.10). If the visibility of a node changes, the set of termination nodes is adjusted – either with a "pull down" (subdivide bounding volume) or a "pull up" (merge bounding volumes) operation. The kd-tree is traversed in a front-to-back order and visible nodes (of previous frames) are rendered immediately during traversal. Also the occlusion query of a termination node is immediately started. The result of an occlusion query is requested later in the traversal. This results in an interleaved rendering and culling scheme, which reduces the latency of the hardware for the occlusion queries. The algorithm is very useful for static scenes and slow camera movements. On the other hand, dynamic scenes and fast changes of the viewpoint are problematic, because the termination nodes will change very often, which reduces the occlusion efficiency. But this is the case for all algorithms, which try to exploit temporal coherence.

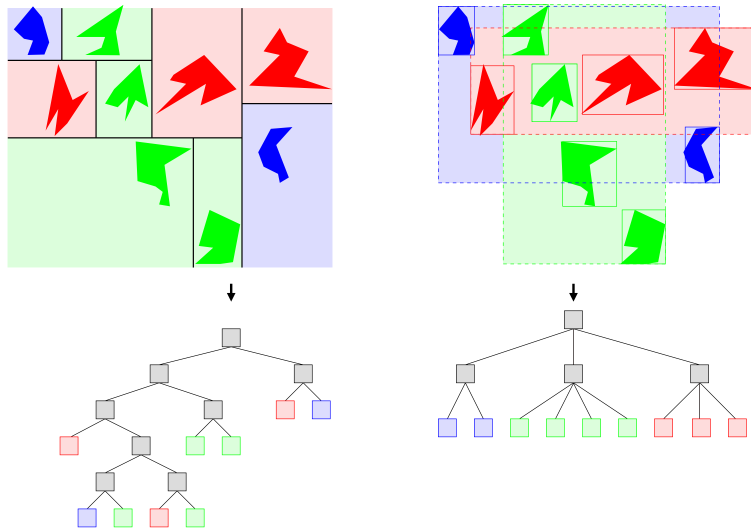

Figure 3.11: Differences between a kd-tree (left) and a scene graph (right). The bounding volumes of a kd-tree are obviously more spatially correlated than the bounding volumes of a scene graph.

The authors claim that the algorithm will work on arbitrary hierarchical data structures for scenes. This is true for spatial structures like octrees and BSP-trees, but not for scene graphs, where the bounding volumes of inner nodes are not strongly spatial correlated (see Figure 3.11). There is much less spatial coherence in a scene graph which will result in a lot of visible occlusion query results after a "pull-up" operation, which significantly reduces the occlusion query efficiency.

Chapter 4

Occlusion Query Implementations

The main focus in this thesis is on real-time occlusion culling for arbitrary scenes in a scene graph environment without precomputing. Also a lot of different types of graphics hardware should be usable, even though more hardware with special support for visibility determination becomes available.

In this Chapter, different approaches to determine the visibility of a scene graph object are compared.

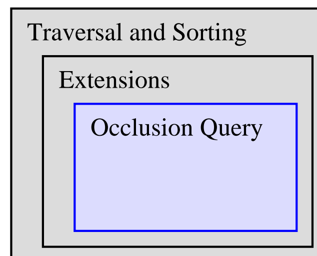

Figure 4.1: Overview of the software architecture. This Chapter describes different implementations of the central occlusion query.

4.1 Introduction

The base algorithms [6, 11] provide some generic, image-space algorithms to get from-point visibility information of a given object. All of them are using the OpenGL graphics pipeline in some way to get the information. The algorithms implement the `isOccluded()` function of Listing 3.2.

The main idea of the following approaches is to exploit the actual z-buffer values

of the OpenGL graphics system. The occlusion test is initialized (for example, disabling z-buffer writes), then occlusion queries can be performed (each request gets an index) and after all queries the results can be requested with the corresponding index. There is no restriction in the geometry for the occlusion test, however, the presented implementation uses the bounding boxes provided by the scene graph for the tests. No precomputing to get a special hierarchy or special data structure is needed, thus arbitrary and dynamic scenes are also supported. Only an up-to-date bounding box hierarchy for each frame is needed.

4.2 Using the OpenGL Depth Buffer

Obviously the OpenGL z-buffer itself can be used to get the visibility information of a bounding volume or object, since it always holds the up-to-date and correct depth-value for every pixel during rendering. To test occlusion, the depth-values of the test geometry are computed with a software rasterizer and compared with the values of the OpenGL z-buffer. The needed `glReadPixels()` to read the OpenGL z-buffer is quite expensive, hence the algorithm is using a caching scheme with a lazy update of fragments of the z-buffer. Each fragment has the same size (for example, 32×32 Pixels), which is a multiple of the data-bus width of the graphics hardware and starts at a memory aligned position to avoid shifting of data. All fragments together create a full resolution z-buffer in software; no hierarchy or lower resolution is used to avoid additional calculations.

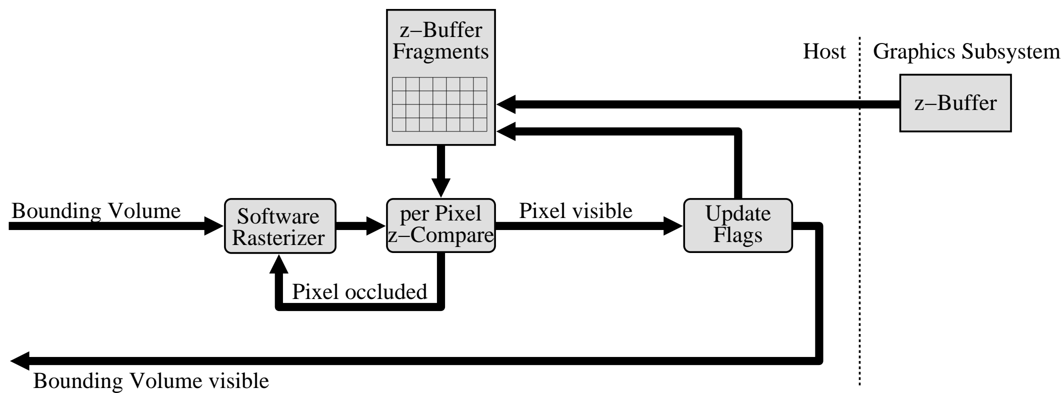

Figure 4.2: Block diagram of the OpenGL z-Buffer read caching scheme.

Two flags for each fragment are used to support a lazy update, an *invalid* and an *unused* flag. At the beginning of every frame, all the *unused* flags are true, because there is no rendered geometry, which could affect the z-values. A tested pixel

against unused fragments leads always to a visible pixel without reading the OpenGL z-buffer. If a pixel is visible in the software rasterizer, the *invalid* bits of the corresponding fragment is enabled, because the actual geometry of the corresponding test geometry will be rendered with OpenGL and the content of the z-buffer may change. For a pixel inside a fragment with a true *invalid* flag, the z-buffer is read to update the fragment and the *invalid* flag is disabled.

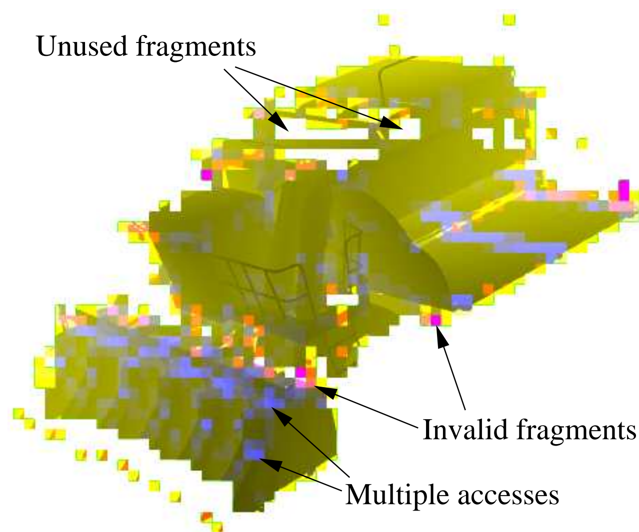

Figure 4.3: Image of the assembled z-buffer fragments after all occlusion queries; not every fragment is needed to reconstruct the visibility of a bounding box. The brightness of a fragment describes, how often the corresponding fragment was read from the OpenGL z-buffer.

4.3 Using the OpenGL Stencil Buffer

Bartz et al. [15] described a technique that the OpenGL stencil buffer can be used to compute visibility informations. The approach works as follows; during rasterization writing to the frame- and z-buffer is disabled. For each pixel of the bounding volume the z-buffer test is applied. If the pixel would be visible, a value is written to the stencil buffer (see Figure 4.4) by using `glStencilOp()`. After rasterizing the bounding volume, the stencil buffer is read and sampled by the application. Occluded bounding volumes will not contribute to the z-buffer, hence will not cause a respective entry in the stencil buffer. On the other hand, partly visible bounding volumes modify the stencil-values of the corresponding, visible pixels.

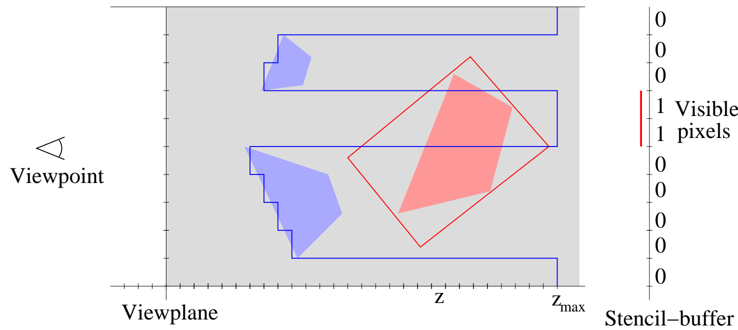

Figure 4.4: Occlusion test with the stencil buffer.

The actual implementation reads the whole region of the covered zone by the bounding volume. This could be optimized with a lazy update like the fragments in Section 4.2 or with the interleaving scanning scheme from Bartz et al. [15]. Multiple queries are possible, if the stencil buffer supports more than one bit. The amount of visible pixels can be also counted, but usually the test is stopped after the first or a necessary amount of visible pixels are found. For more details please refer to [15].

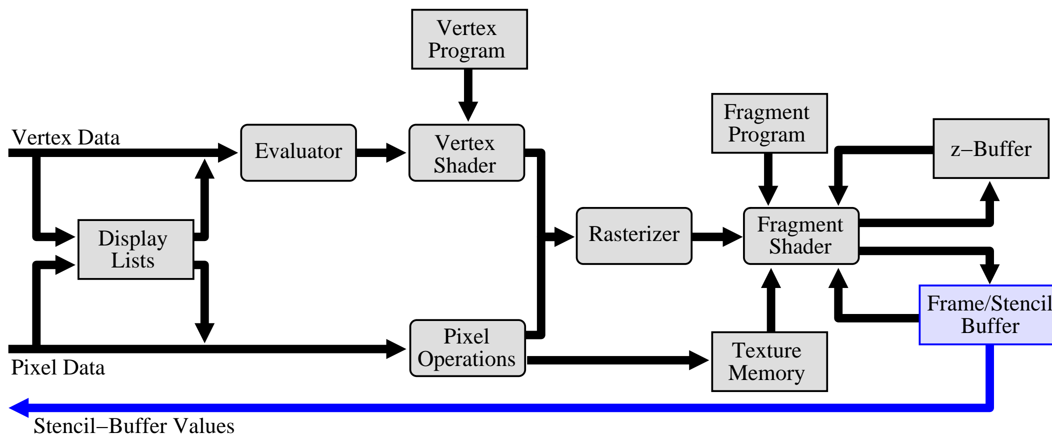

Figure 4.5: Block diagram of the OpenGL stencil buffer read.

4.4 OpenGL Extensions for Occlusion Culling

As presented in Section 3.3, OpenGL extensions can be used to get the occlusion information. A small application was used to measure the latency of the extensions on the graphics hardware. Test boxes with different sizes were moved in different

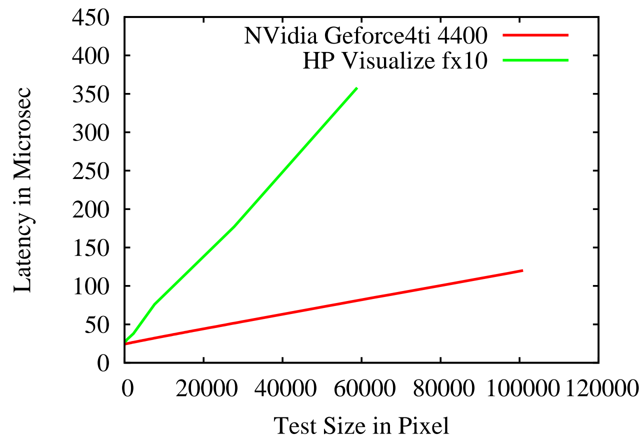

Figure 4.6: Latency of the HP Occlusion Flag on an Intel P4@2400MHz with a NVidia GeForce4Ti 4400 (red) and on an Intel P3@750MHz with a HP Visualize fx10 (green). Both running Linux.

frames from completely visible over partly visible to completely occluded. The performance of the hardware extensions depends on the fill-rate of the z-buffer. Larger geometry needs more time for the test, because the whole geometry passes always the z-buffer stage of the rendering pipeline. Figure 4.6 shows the correlation between the size of the geometry in screen-space and the latency for an occlusion test request. With enabled backface culling, the test is almost twice as fast as without, because with backface culling, only one scan through the z-buffer for the front-face is done. The graphics hardware rasters always the complete bounding volume, but the rasterization could be stopped after the first visible pixel when using the HP extension. With the NVidia extension, the hardware has to raster always the whole bounding volume to determine the full amount of visible pixels. This is a drawback, especially for larger bounding volumes. In addition, a more complex hardware implementation could use a hierarchical representation of the hardware z-buffer, which would also speed up the occlusion tests.

4.5 Evaluation

To compare the different techniques the Formula One Car model from the Jupiter project was used. The model has about 750.000 polygons in 306 geometry nodes (see Section 8.2.2). A camera path with 342 frames was created; In every frame the whole model is located within the viewing frustum, therefore view frustum culling

itself does not remove geometry. In Figure 4.7 and Table 4.5, the resulting frame rates are shown for the different occlusion culling techniques. A simple depth first traversal of the scene graph with an additional front-to-back sorting of the geometry was used [6].

	Avg. fps	Deviation fps	Min. fps	Max. fps	Avg. speedup
No occlusion culling	3.77	0.03	3.58	3.85	0.0%
Stencil test	4.28	0.23	3.46	5.13	12.0%
Z-Buffer test	4.42	0.28	3.65	5.15	14.8%
HP Flag	5.70	0.41	4.44	6.67	33.8%

Table 4.1: Comparison between different occlusion query implementations [6].

The first benchmark (no culling) shows the performance of OpenSG without any occlusion queries. In the second benchmark, the performance of the stencil buffer test was evaluated. In the third test, the z-buffer technique was applied and in the last one the HP Occlusion Flag was used. For all benchmarks a Dual Pentium III with 750 MHz with a HP VISUALIZE fx10 running Linux was used for rendering. The resulting frame rates show average speed-ups between 12% and 34%.

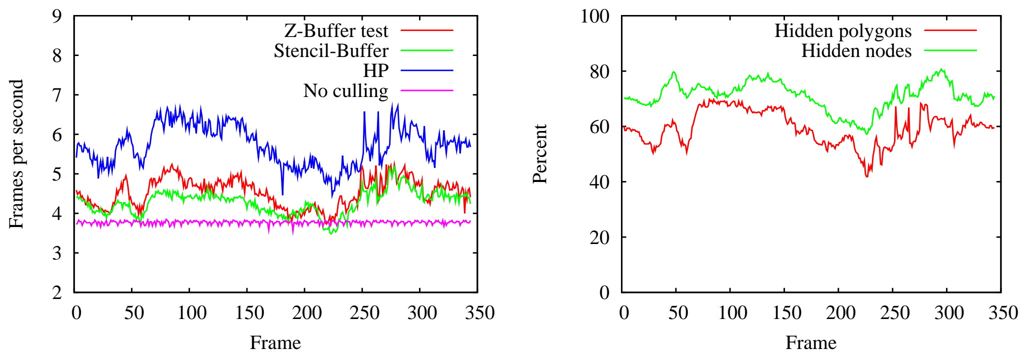

Figure 4.7: Left: Frame rates for the camera path and right: percentage of occluded nodes and polygons.

Occlusion culling generally depends on the scene and its depth complexity. Figure 4.7 (right) shows the percentage of hidden nodes and polygons in every frame. The limited depth complexity of the test dataset (about 60% of the polygons are detected as occluded) leads only to a limited culling performance. In scenes with a higher depth complexity, a better performance can be expected.

The benchmarks show that the HP Occlusion Flag is the fastest solution in this test. The stencil- and z-buffer-tests show similar results, whereby the stencil-test will perform better in frames with lower depth-complexity (less setup and rasterization time in software), while the z-buffer-test is faster in frames with more depth-complexity, because the z-buffer in software needs less updates (for occluded nodes no update is necessary).

The techniques show differences in the load of the graphics hardware and the host's CPU. With the readback of the z-buffer in Section 4.2 most work is done by the host's CPU, so this approach is very useful for systems with lower graphics hardware capabilities. In addition no state changes on the graphics hardware are needed. The stencil buffer technique of Section 4.3 distributes the load on the host's CPU and the graphics hardware. But state changes are necessary to setup the stencil buffer, anyway this technique is useful on standard OpenGL hardware without special occlusion extensions and lower CPU power. On modern graphics hardware the special extensions are the first choice (see Section 4.4). Most of the work is done by the graphics hardware and the host's CPU can be used for other calculations like sorting. But state changes are necessary, so that the number of occlusion queries should be minimized, which is the focus of the next Chapter.

Chapter 5

Avoiding Predictable Occlusion Queries

The previous presented techniques to get the visibility of an object are using the OpenGL graphics hardware in some way. Although the performance of graphics hardware is continously increasing, each request to the hardware causes some latency. This chapter presents some algorithms, which reduce the number of occlusion queries.

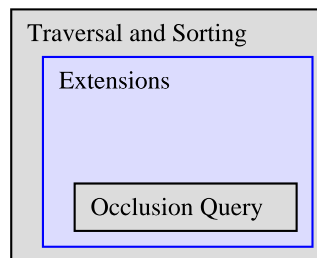

Figure 5.1: Overview of the software architecture. This Chapter describes extensions, to avoid predictable occlusion queries.

5.1 Introduction

To reduce the number of occlusion queries on the graphics hardware, some aspects of the visibility of occluders and occludees are taken into account. A trivial observation shows if there is no rendered geometry in the actual frame, an occlusion query will *always* lead to a visible result, because there is no occluding geometry. The following Sections show further details, how occlusion queries can be avoided, if their result can be predetermined. The techniques are used on top of the occlusion queries of Chapter 4.

5.2 Occupancy Map

During traversal of the scene graph, an algorithm has to decide when to perform an occlusion query for a given node. Especially from viewpoints with low occlusion a lot of occlusion tests are unnecessary, because they return a visible result, so that the rendering of the corresponding geometry gets more expensive. The idea is that only in (occupied) regions with already rendered geometry, an occlusion test makes sense. The *Occupancy Map* [9] is a small data structure which manages occupied regions of the screen space. In not occupied regions an occlusion test will always return visible pixels due to the lack of occluding pixels (see Figure 5.2).

As noted in the previous Section 4.4, the latency of an occlusion query depends on the number of rasterized pixels of the bounding volume. In particular large, partially visible bounding volumes will spent significant time in the rasterization stage of the graphics accelerator, without any benefit for the rendering performance. In order to reduce the associated costs, the approach tries to avoid occlusion queries with a visible result.

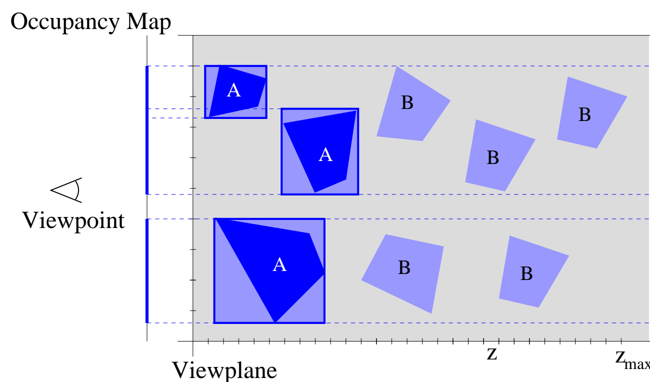

Figure 5.2: Construction of the Occupancy Map; bounding boxes of the geometry in the front (A) are added.

5.2.1 Implementation

The Occupancy Map is realized as a small bit field. Each bit represents an occupied or unoccupied region of the screen-space. Storing depth values or other information is not necessary, if the requests occur in a partial depth sorted order. Due to performance reasons, the Occupancy Map is updated with the screen-space bounding boxes of the rendered geometry, which is not exact, but a conservative approximation of the occupied regions.

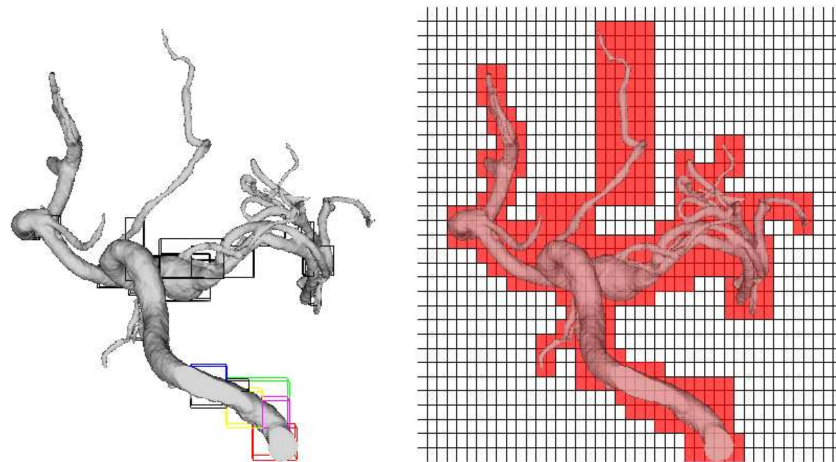

Figure 5.3: *Left:* Scene with low occlusion. *Right:* Occupancy Map for scene [9].

A first implementation [5, 9] was done with the Jupiter scene graph and later the approach was added to the OpenSG system [7].

Before a node's visibility is tested with an occlusion query, its screen-space bounding box is compared with the Occupancy Map values. As soon as the Occupancy Map detects that the target screen area of the node's bounding box is partial "empty", it cancels the occlusion query and initiates the rendering or traversal of the respective scene graph node (see Section 6.4 – Stage 1). Note that the Occupancy Map is conservative, since it is essentially storing the lower resolution coverage information of the framebuffer. However, it is not exact and will occasionally initiate the rendering of geometry which would have been determined occluded by the actual occlusion query. This is due to the approximation of the scene entity bounding box by a screen space axis-aligned bounding box (AABB). Nevertheless, tests found that with the used Occupancy Map size (see below), this did not turn up to be a problem [5, 9].

Technically, an Occupancy Map is a cache optimized bit-field realized as an array of 32 Bit integers with 256 entries (to fit into a cache line). Every bit represents a tile in the screen space. If a bit is set, the respective tile is occupied by already rendered geometry. Otherwise, no geometry has yet been rendered into the associated screen region (see Fig. 5.3). The effectiveness of the tile size is a trade-off between precision (resolution) and overhead (memory and update). Meißner et al. provided interesting measurements on the effective resolution in the context of a hardware implementation [42]. In our context, an Occupancy Map size of 1024 Bytes is quite effective. The representing tile size can be adapted to the window size; for example, in a window of 1024×768 pixels every tile represents 4×24 pixels. If the whole

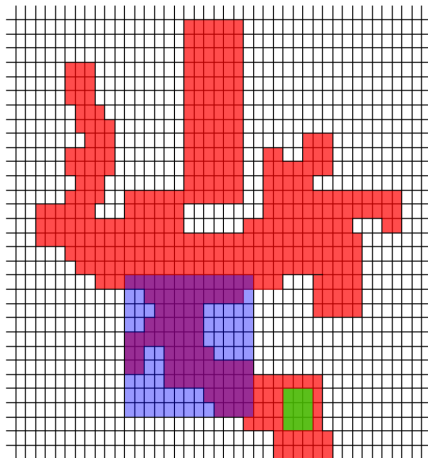

Figure 5.4: Request to the Occupancy Map; the blue box is detected as “not occluded”, the green one as possibly occluded. For the latter one, an occlusion query will follow.

scene is inside the view-frustum, the screen space size of the scene’s bounding box can be used to scale the Occupancy Map to enable a better resolution. Due to the small size, the Occupancy Map can be easily accommodated in the first level cache of the CPU to permit an extremely low latency (orders of magnitude lower than a read back from the graphics accelerator to acquire the occlusion query result).

For a lookup, the screen space AABB of the corresponding node is checked whether it overlaps with empty (unset) regions of the Occupancy Map. If that is the case (blue box in Fig. 5.4), the corresponding AABB is assumed visible and the occlusion query is canceled. Note that for good performance, the test nodes have to be organized partially front-to-back.

5.3 Additional Depth Buffer in Software

In contrast to the Occupancy Map of the previous Section, the presented *Software Depth Buffer* focuses on occluded bounding volumes.

For scenes with high depth complexity, occlusion tests can be saved by a software implementation of a z-buffer. Rendering of the scene geometry in software is too expensive, but occluded bounding boxes can be used as an approximation (see Figure 5.5). Thus the bounding boxes of previous (hardware accelerated) occlusion queries are rendered into the Software Depth Buffer. Before another bounding volume is tested by the occlusion query, it can be tested with the Software Depth Buffer. Fill-rate and rasterization calculations for the Software Depth Buffer can be saved by

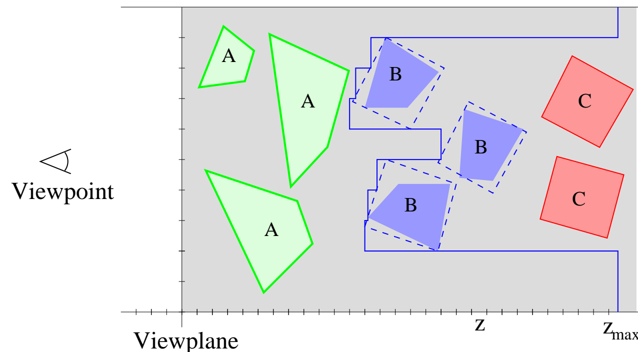

Figure 5.5: Occlusion test with the Software Depth Buffer; (A) visible geometry, (B) tested with an OpenGL test, (C) tested by the software approach.

a lower resolution than the depth buffer of the graphics subsystem. Also if the application knows occluders, they can be rendered into the Software Depth Buffer before starting any other tests.

Unfortunately, bounding boxes of scene graph nodes can intersect each other in object space so that the Software Depth Buffer must save depth values in contrast to the Occupancy Map, due to an overlap test is not being sufficient to calculate conservative visibility. In an octree or BSP tree environment it is possible to realize a strict front-to-back rendering. Therefore the Software Depth Buffer could be implemented as a simple "bitfield", which would further significantly speedup the test.

5.4 Temporal Coherence

To reduce the number of occlusion queries, temporal coherence between frames can be used. Based on the assumption that a visible scene graph node in one frame is likely to be visible in the next frame, occlusion queries for these nodes can be saved. In our implementation [11], the result of an occlusion query is stored for the corresponding node. Occlusion queries for a visible node in frame n are not performed in the following $n+1\dots d$ frames. The value d changes between 2 and 4 to balance the number of occlusion queries over all frames. If d is constant, a lot of queries are performed in frame n, while in frames $n+1\dots d$ these are saved. Experiments showed that this would result in very high frame rate jitter.

For aggressive occlusion culling with a small reduction in image quality, occluded nodes can also be assumed as occluded in the next, following frames. This leads to a non-conservative, aggressive approach, since failure of this assumptions

will result in missing geometry in the image. In our implementation, it is assumed that an occluded node is only occluded in the next two frames to minimize these image errors while moving the camera. Furthermore a preliminary test with the Occupancy Map is performed to avoid errors of large visible nodes in the front of the scene.

Temporal coherence can be very problematic in dynamic scenes or if the viewpoint is changing fast. Bittner et al. [16] presented another temporal coherence algorithm, but they also benefit from a balanced hierarchy of the scene in a kd-tree (object-space coherence). Also the problem of dynamic scenes is not addressed by their approach. Due to the fact that the presented approach is a heuristic, the implementation is unrivaled. The “magic” numbers, how long a visibility result is valid has an influence to the resulting frame rates, which makes comparison between different models and views difficult.

5.5 Results

All tests in this Section were performed with the OSGViewer application [6] and an OpenSG scene graph. A PC with AMD Athlon 2500 XP (1.8 GHz) running Linux and a NVidia Geforce3 was used as platform. During traversal, hierarchical occlusion culling with the OpenGL extension from NVidia in conjunction with the previous extensions was applied. The traversal of the scene graph occurred in an object-space front-to-back order with applied hierarchical culling. The nearest corner of a node’s bounding box was used as criteria for the ordering and the nodes’ bounding boxes were used for the occlusion queries. All frames were rendered at a high resolution of 1140×755 with 24 bits color depth. The Software Depth Buffer had a quarter resolution of the viewport if used for the tests.

	Number of polygons
Cotton Picker	10 610 166
Formula One Car	746 827
City	4 056 195

Table 5.1: Test models.

First, the Cotton Picker model (see Table 5.1 and Section 8.2.1) and a camera path with 280 different frames was used. In the right Figure 5.6 the amount of visible and occluded polygons (detected by a bounding box occlusion test) for each frame is shown. Approximately between frame 150 and frame 250 the view frustum culler

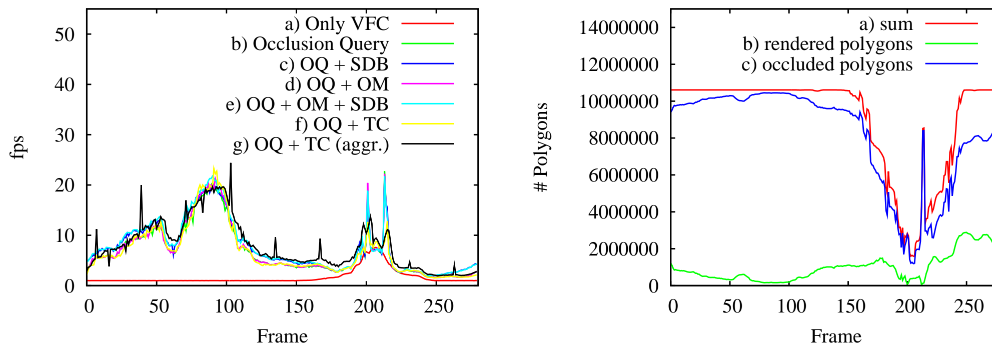

Figure 5.6: Performance (left) and visible/occluded polygons (right) during rendering of the Cotton Picker camera path.

	Avg. fps	Dev fps	Min. fps	Max. fps	Speedup
No occlusion culling	1.7	1.5	1.0	7.6	0%
OpenGL Occlusion Query	6.7	4.8	1.6	22.7	294%
OpenGL OQ + Occupancy Map	6.9	5.0	1.6	22.7	306%
OpenGL OQ + Software Depth Buffer	7.5	4.8	1.9	21.7	341%
OpenGL OQ + OM + SDB	7.7	5.1	2.0	21.7	353%
OpenGL OQ + Temporal Coherence	6.7	5.0	1.5	23.3	294%
OpenGL OQ + TC (aggressive)	7.9	4.9	1.9	24.4	364%

Table 5.2: Comparison of the performance timings for the Cotton Picker model.

removes some nodes. Table 5.2 concludes some average performance results. Each extension is able to speed up the average rendering speed. The fastest results are with enabled Occupancy Map and Software Depth Buffer, where the average rendering speed increases by about one fps, which is a speedup of 15% compared to using only the OpenGL occlusion query.

A Formula One Car was the second test model (see Section 8.2.2). A camera path with 270 different frames was rendered to compare the performance. Like the Cotton Picker path, in some frames parts of the scene are outside the view frustum. Right Figure 5.7 shows the visible and occluded polygons during the camera path. The rendering performance can be found in the left Figure 5.7. Table 5.3 concludes the timings for the Formula One Car rendering. Similar to the results of the Cotton Picker model the rendering gains approximately one frame with the extensions, but

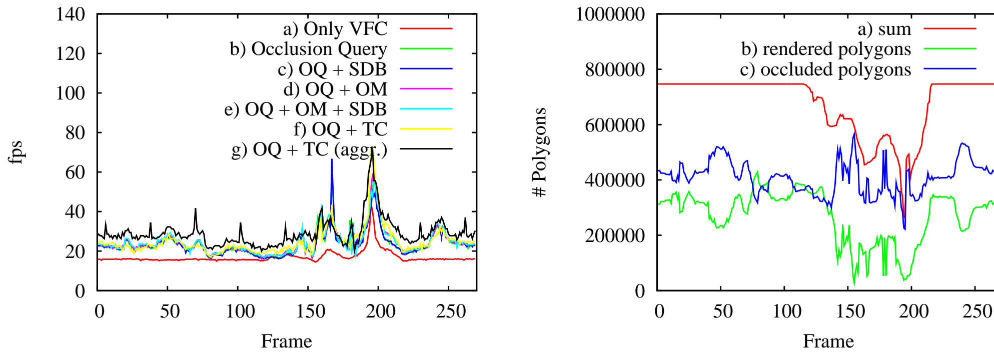

Figure 5.7: Performance (left) and visible/occluded polygons (right) during rendering of the Formula One Car camera path.

	Avg. fps	Dev fps	Min. fps	Max. fps	Speedup
No occlusion culling	16.9	3.3	14.5	43.5	0%
OpenGL Occlusion Query	23.6	6.5	16.4	58.8	40%
OpenGL OQ + Occupancy Map	24.4	6.5	16.1	58.8	44%
OpenGL OQ + Software Depth Buffer	23.7	6.4	16.1	66.7	40%
OpenGL OQ + OM + SDB	24.5	6.3	16.7	55.6	45%
OpenGL OQ + Temporal Coherence	25.0	7.4	16.9	71.4	48%
OpenGL OQ + TC (aggressive)	29.0	7.1	20.0	71.4	72%

Table 5.3: Comparison of the performance timings for the Formula One Car model.

in contrast to the Cotton Picker, this is just a speedup of 4%. Due to the lower complexity of the model the number of occlusion queries is lower, which results in a reduced dependency of the occlusion query performance during rendering.

In a last test, a City model with some Formula One Car models in the streets was used for rendering (see Section 8.2.3). The camera path consisted of 110 frames and also in some frames parts of the scene were outside the view frustum (see Figure 5.8 left). In contrast to the Formula One Car, the City model is much more complex and in many frames a lot of geometry is occluded. Table 5.4 and right Figure 5.8 show the rendering performance. In addition to the occlusion queries, the average performance increases by 1.5 fps with the Occupancy Map and the Software Depth Buffer, which is equal to a speedup of 11%.

The temporal coherence algorithm shows only with the Formula One Car model

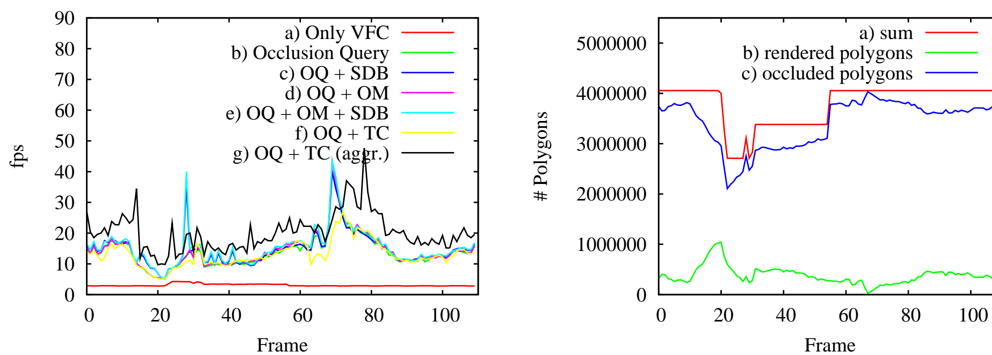

Figure 5.8: Performance (left) and visible/occluded polygons (right) during rendering of the City model.

	Avg. fps	Dev fps	Min. fps	Max. fps	Speedup
No occlusion culling	3.1	0.4	2.8	4.3	0%
OpenGL Occlusion Query	13.8	5.2	5.1	40.0	345%
OpenGL OQ + Occupancy Map	14.3	5.7	5.1	43.5	361%
OpenGL OQ + Software Depth Buffer	14.7	5.4	5.3	40.0	374%
OpenGL OQ + OM + SDB	15.3	5.9	5.3	43.5	393%
OpenGL OQ + Temporal Coherence	13.1	4.1	4.9	27.0	322%
OpenGL OQ + TC (aggressive)	20.1	6.1	9.7	47.6	548%

Table 5.4: Comparison of the performance timings for the City model.

and with the aggressive technique speedups compared with the standard occlusion queries. However, the conservative technique shows a slowdown applied to the City model and the aggressive technique generates image errors, therefore the presented temporal coherence algorithm is not useful in a commonly usable scene graph environment.

5.5.1 Occupancy Map and Software Depth Buffer

To measure, how many occlusion queries can be saved, the amount of visible and occluded occlusion queries were counted with and without the additional techniques. Figures 5.9, 5.10, and 5.11 show the percentage of the savings of the Occupancy Map and the Software Depth Buffer from all occlusion tests in the frames of our camera paths. In addition, Table 5.5 shows the averaged values.

For the Cotton Picker the Occupancy Map saves 31% of occlusion tests with a visible result and the Software Depth Buffer saves 73% of occlusion tests with an occluded result. Of course, these values are lower in scenes with lower depth complexity.

The Occupancy Map saves only 17% of occlusion tests with a visible result and the Software Depth Buffer saves 55% of occlusion tests with an occluded result for the Formula One Car model. The lower value of the Occupancy Map results from many, partly visible, small objects, which are not in the front of the model.

For the City model the Occupancy Map saves 17% of occlusion tests with a visible result and the Software Depth Buffer saves 42% of occlusion tests with an occluded result.

	Avg. # Queries Visible	Avg. Saving	Avg. # Queries Occluded	Avg. Saving
Cotton Picker:				
Only Occlusion Queries	1251		2205	
With OM + SDB	868	31%	587	73%
Formula One Car:				
Only Occlusion Queries	80		186	
With OM + SDB	66	17%	83	55%
City:				
Only Occlusion Queries	174		582	
With OQ + OM + SDB	145	17%	327	42%

Table 5.5: Comparison of the number of occlusion queries for the Cotton Picker, the Formula One Car, and the City model.

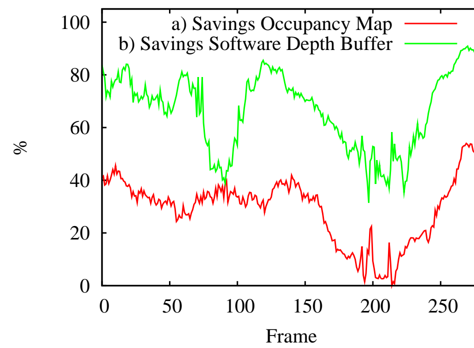

Figure 5.9: Savings of extra occlusion tests by the Occupancy Map and the Software Depth Buffer for the Cotton Picker model.

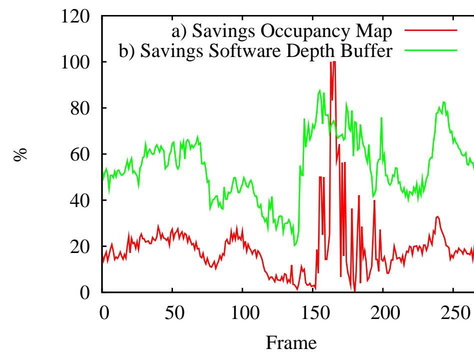

Figure 5.10: Savings of extra occlusion tests by the Occupancy Map and the Software Depth Buffer for the Formula One Car model.

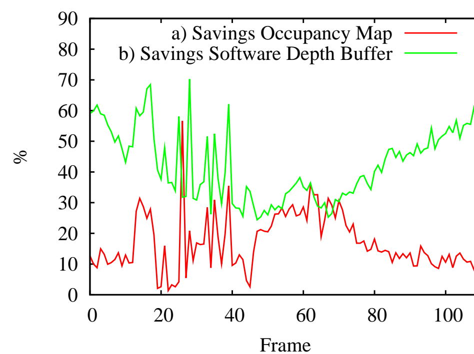

Figure 5.11: Savings of extra occlusion tests by the Occupancy Map and the Software Depth Buffer for the City model.

Chapter 6

Occlusion Driven Traversal

The previous Chapter described some techniques to reduce the number of occlusion queries to reduce the load of the graphics hardware. Another way to make occlusion culling more efficient is to shadow the latency of occlusion queries with other work on the graphics hardware.

This Chapter describes a special traversal technique of the scene graph to reduce the occlusion query overhead.

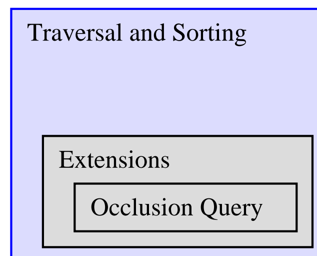

Figure 6.1: Overview of the software architecture. This Chapter describes a special traversal and sorting technique of the scene graph with focus on efficient use of occlusion queries.

6.1 Introduction

As noted in previous Chapters, the image space occlusion queries of graphics hardware require state changes and rasterization bandwidth (for the testing geometry). Therefore, this thesis introduces a novel hierarchical traversal technique for scene graphs, which significantly reduces the latency of occlusion queries. The approach

reduces the number of state changes by using *multiple* occlusion queries and allows the *hierarchical* culling of occluded subtrees from the scene graph. The overall goal of the approach is a stable render performance improvement for a varying set of models and viewing situations, even if little or no occlusion is present, where regular occlusion culling methods have high overhead costs with little benefits. The presented traversal technique requires no preprocessing or special (spatial) data structures. Finally, the approach supports dynamic and animated scenes, if the scene graph is updated accordingly.

6.2 Occlusion Query

To obtain occlusion information, the mentioned (see Section 3.3 and 4.4) hardware supported occlusion queries from NVidia are used. For an occlusion test, the test geometry is rendered in the occlusion query mode with disabled frame and z-buffer writes to avoid actual modifications to the framebuffers. Two different costs are associated with an occlusion query; first it has to wait for the completion of the state changes (because of disabling of frame and z-buffer writes and other state changes) and second it rasterizes the test geometry.

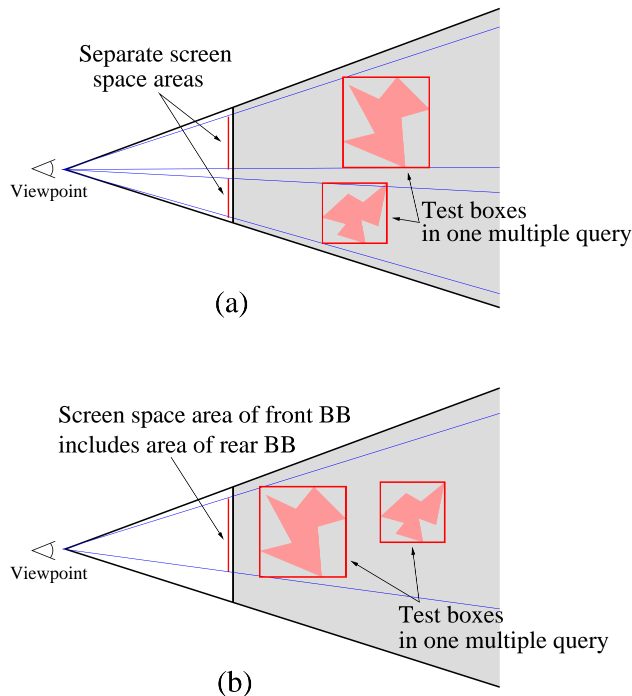

Figure 6.2: Multiple query without (a) and with (b) possibly redundant results.

The NVidia extension supports clustering of multiple tests in one query to reduce the latency of setup costs for an occlusion query. This way, the visibility of more than one bounding volume can be determined at the same time, while intermediate visibility results are not guaranteed to be considered. For each tested bounding volume, the corresponding occlusion result is asynchronously returned (see Figure 3.6 of Section 3.3). A major problem of multiple queries is the selection of the bounding volumes. Because intermediate results are not taken into account. Two each other occluding bounding volumes may return a false positive result (both not occluded, see Figure 6.2); the second volume located behind the first one is tested against the not up-to-date z-buffer, since the geometry of the first bounding volume is not yet rendered. To avoid this problem, the presented approach uses a lower-resolution representation of projected bounding boxes to utilize the screen-space coherence.

6.3 Organization of Multiple Occlusion Queries

As mentioned, the latency of an occlusion query can be reduced by using multiple occlusion queries. This method reduces setup costs, because state changes are solely necessary before and after the multiple query and the results can be collected asynchronously (see Section 3.3). However, redundant queries (see Figure 6.2) have to be avoided in order to circumvent false-positive results of geometry that is indeed occluded. To address this problem, the algorithm uses an Occupancy Map for each multiple query. In contrast to [9], an hierarchical approach exploiting multiple occlusion queries is presented.

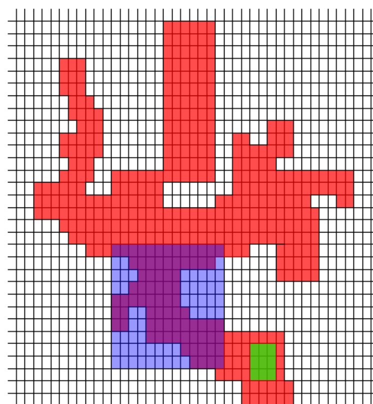

Figure 6.3: Request to the Occupancy Map; the blue box is detected as “not occluded”, the green one as possibly occluded. For the latter one, an occlusion query will follow.

In contrast to Section 5.2, the Occupancy Map has a slightly different meaning. A covered area in such an Occupancy Map means that this region is covered by a bounding box from the corresponding test list. Only if at least one bit in the Occupancy Map is not set in the area of the corresponding bounding box, the bounding box is added to a multiple occlusion query (see Figure 6.3). This means that the bounding box will test a region in screen-space, which is not yet covered by another bounding box from the respective test list. Bounding boxes can overlap in screen-space, which could result in redundant queries, but they never occlude each other. If all Occupancy Map bits covered by the bounding box are already set, the bounding box is tested in a subsequent Occupancy Map. A bounding box, which is added to an occlusion query, is always rendered into the tested Occupancy Map to mark the corresponding screen space region as used.

6.4 Traversal

In this approach, hierarchical view-frustum and occlusion culling of the nodes in the scene graph is applied. While the inner nodes of the scene graph only contain the bounding volume of their associated sub-tree, the leaf nodes contain the actual geometry and their corresponding bounding boxes. The experiments are performed with OpenSG.

Interleaved culling and rendering is performed during the scene graph traversal. Two priority queues (*traversal* and *pending* queue) are used to organize the traversal and the respective multiple occlusion queries. The priority queues are functioning in a *double-buffered* manner. Only one Occupancy Map is used in this approach, and it is cleared after each multiple occlusion query. The primary algorithm, which we presented in [9], always used five instances of Occupancy Maps without hierarchical culling.

To restrict the maximum number of multiple occlusion queries per frame, the Occupancy Map is used for only o multiple queries. All remaining nodes are processed by a brute force applying of occlusion queries. Limited temporal coherence is exploited at this point, since o is calculated from frame to frame: $o_{frame+1} = 7/10 \cdot m_{frame}$, where m is the number of all (organized by Occupancy Maps + remaining queries) multiple occlusion queries. Note that even for drastic movements – which destroy temporal coherence –, the method is still conservative. Only the efficiency might be reduced, due to the occlusion queries of the last stage.

The traversal technique can be partitioned into three main stages; starting with the root node, it takes the bounding volume of the node and performs a view-frustum culling test. If the node (actually its bounding volume) is determined inside the view-

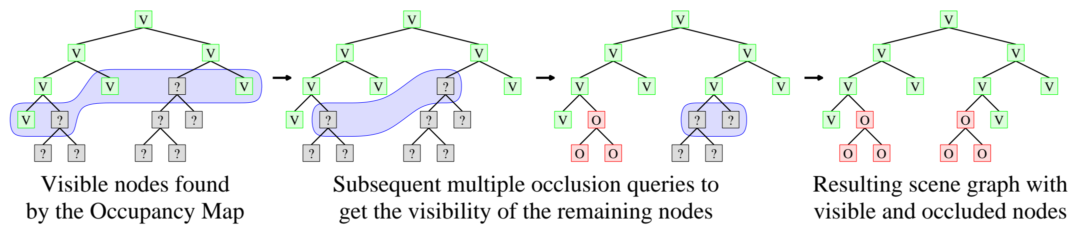

Figure 6.4: Traversal of the scene graph.

frustum, its child nodes are added to a front-to-back sorted queue (*traversal* queue T in Figure 6.5) of current nodes. Otherwise, the whole sub-tree is skipped. The closest corner of the node's bounding box to the current viewpoint is used for the front-to-back sorting.

In the *first stage*, the Occupancy Map is used to find the visible nodes in front of the scene (equal to Section 5.2):

```
// Listing 6.1 – Stage 1

while(traversal queue T not empty)
{
  node = get first node of traversal queue T;
  if (node is outside view frustum)
    cull node;
  else
  {
    if (node is visible in occupancy map)
    {
      if(node is geometry)
        render and assign node to occupancy map;
      else
      {
        add children front-to-back
                 to traversal queue T;
      }
    }
    else
      add node to pending queue P;
  }
}
```

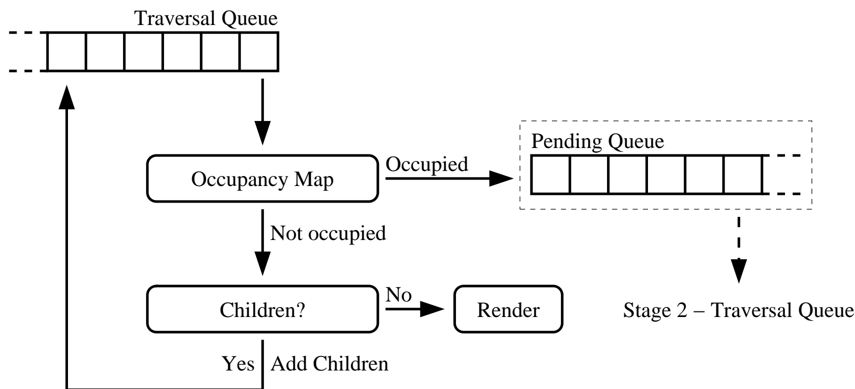

Figure 6.5: Stage 1 – Request to the Occupancy Map; Leaf (geometry) nodes are directly rendered.

Nodes with a bounding box visible in the Occupancy Map, are assumed visible in the scene, since there is no rendered geometry up to now which covers the associated region. If such a node is a geometry node, it is rendered and the Occupancy Map is updated by its screen space bounding box. Nodes, whose bounding box is covered in the Occupancy Map are added to the *pending* queue P. These nodes are probably occluded and therefore tested with occlusion queries in the next stage of the traversal scheme. In Fig. 6.5, an overview of the first stage of the traversal is given. If the *traversal* queue T is empty the algorithm proceeds to stage two and the *pending* queue replaces the *traversal* queue, $T := P$.

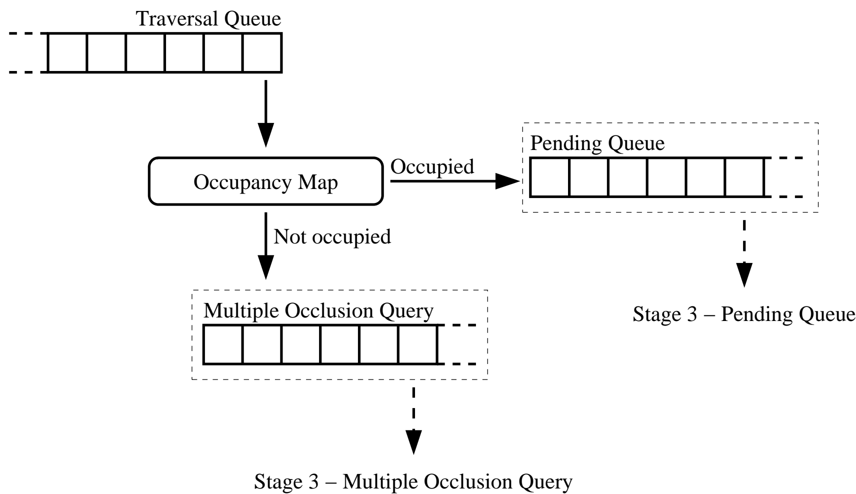

Figure 6.6: Stage 2 – sorting of scene graph nodes during traversal.

In the *second stage* (see Fig. 6.6), multiple occlusion queries are performed on the nodes of the scene graph. To avoid false positive results, an Occupancy Map is used to distribute the occlusion tests in different screen space regions (cp. Fig. 6.2). Nodes, whose bounding box are not yet tested for occlusion, are added to the *pending* queue P for later processing:

```
// Listing 6.2a - Stage 2

m = 0;

while(pending queue P not empty && m<max_tests)
{
  clear occupancy map;

  T = P;

  while(traversal queue T not empty)
  {
    node = get first node from traversal queue T;

    if(node is outside view frustum)
      cull node;
    else
    {
      if(node is visible in occupancy map)
      {
        add node to test list;
        assign node to occupancy map;
      }
      else
        add node to pending queue P;
    }
  }

  perform multiple occlusion query with test list;

// -> 6.2b
```

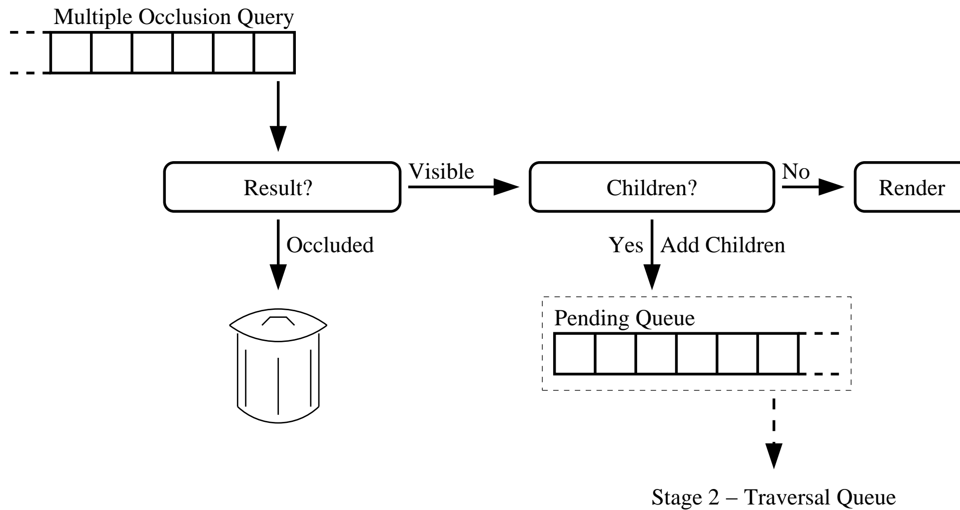

Figure 6.7: Stage 3 – the result of the hardware occlusion query causes culling or further processing.

In the *third stage* (see Fig. 6.7), the results of the multiple occlusion query are collected. Visible nodes are either immediately rendered (geometry (leaf) nodes) or their children (of inner nodes) are added to the *pending* queue P for further processing:

```
// Listing 6.2b - Stage 3

  for(each test node){
    if(node is visible){
      if(node is geometry)
        render node;
      else
        add children to pending queue P;
    }
    else
      cull node;
  }

  m++;
}
```

Stage two and three are *looped* (after stage three the *pending* queue replaces the *traversal* queue: $T := P$), until all nodes from the *traversal* queue T are processed or a maximum number of organized multiple occlusion queries is reached.

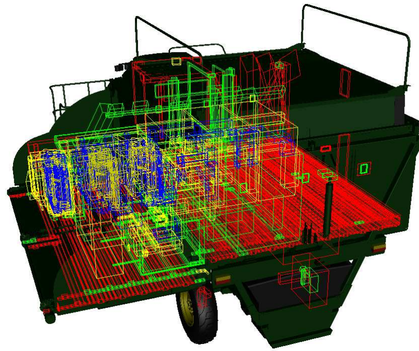

Figure 6.8: Occluded bounding boxes for a given view of the Cotton Picker model. All bounding boxes of an individual multiple occlusion query have the same color.

To avoid too many occlusion queries given by the Occupancy Map in the backstage of the scene, brute force multiple occlusion queries on the remaining nodes are performed, since bounding boxes in the back are usually occluded:

```
// Listing 6.3

while(pending queue P not empty){
  perform multiple occlusion query with pending queue;

  clear pending queue;

  for(each visible node){
    if(node is geometry)
      render node;
    else
      add children to pending queue P;
  }

  m++;
}

max_tests = 7/10 * m;
```

Overall, the presented traversal technique provides hierarchical culling, since complete subgraphs are culled, if the bounding box of the subgraph is occluded. Figure 6.4 demonstrates the traversal in the scene graph and Figure 6.8 shows the mapping of the bounding boxes to a corresponding multiple occlusion query.

6.4.1 Complexity

Recapitulating: The simplest way to render a scene graph is with a stack-based depth-first traversal of the scene graph. This kind of traversal has linear complexity, but is very inflexible and does not care about the rendering performance. A much more flexible scheme can be realized by using a priority-driven traversal of the graph. By using specific priorities (for example: screen-space size, distance to the view-point, or material parameters such as transparency) different orders for rendering can be achieved. Due to the sorting of the needed priority queue, the complexity is $O(n\log(n))$ (sometimes this can be optimized by using a hash-table). Unfortunately, the presented algorithm uses two priority queues, which are working in an interleaved fashion. This leads to a worst case complexity of $O(o \times n\log(n))$ (Figure 6.9 gives an example of a worst case situation), where o is the number of used Occupancy Maps. Fortunately, this is a very rare situation because all bounding boxes have to overlap each other in screen-space *and* they must be from different subgraphs.

Another problem in the presented implementation is the repeated sorting of the nodes into the priority queues due to the double-buffered scheme. This can be avoided by using only one priority queue with random access. Therefore, an algorithm can remove occluded nodes and add children of visible nodes without copying older nodes.

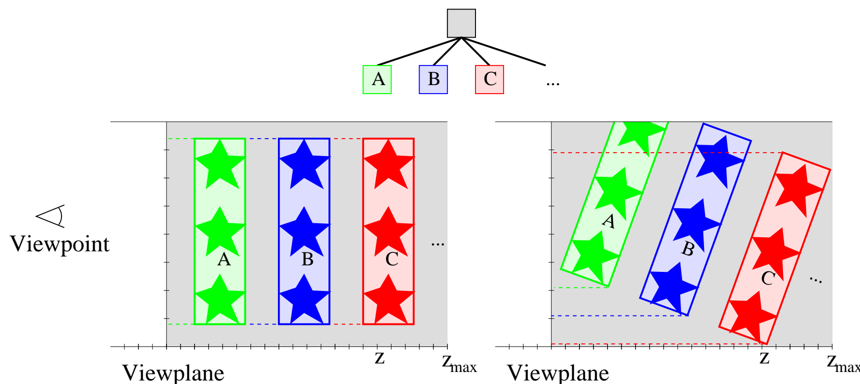

Figure 6.9: Two different situations for the traversal performance. Left: worst case situation, the bounding boxes are overlapping each other and they are from different subgraphs. Right: The bounding boxes do not completly overlap each other and the algorithm can merge the occlusion queries of the different bounding boxes in a multiple occlusion query.

6.5 Results

To evaluate the performance of the multiple occlusion queries organized by the Occupancy Maps, an Intel P4@2400 MHz with a NVidia Geforce FX5600XT and the models listed in Table 6.1 were used. A camera path for each model at a resolution of 800×600 and 32 Bit color depth was rendered. The occlusion culling was done with the NVidia occlusion query extension.

	Number of polygons
Boom Box	644 268
Formula One Car	746 827
F1 Animation	2 242 481
Cotton Picker	10 610 166
Big City	64 898 464

Table 6.1: Test models.

The Boom Box, the Formula One Car, and the Cotton Pickerare MCAD models (see Section 8.2.4, 8.2.2 and 8.2.1). The Big City model is an artificial city with some Formula One Car models in the streets and has the highest complexity of the four models with high depth complexity (see Section 8.2.3). F1 Animation is a simple animation with three Formula One Car models as test for dynamic scenes. The cars driving an "8" behind each other around two obstacles. We did not optimize the scene graphs for occlusion culling or traversal.

We rendered different camera paths for each model. In some frames, the camera zoomed into the scene and parts of the scene outside of the view frustum were culled. Our measurements are performed a) only with view frustum culling, b) with a single synchronous occlusion query for each node in the scene graph (without using an Occupancy Map), c) with multiple occlusion queries organized by the previously described traversal algorithm.

With occlusion culling, average frame rates between 11.1 fps (F1 Animation) and 33.7 fps (Formula One Car) for the different models (see Table 6.2) can be achieved. With the new traversal scheme with Occupancy Maps and multiple occlusion queries the new traversal algorithm improved these results to 16.5 fps (F1 Animation) and 42.9 fps (Boom Box). The best speed-ups were achieved with multiple occlusion queries in scenes with lower depth complexity, because of the trade-off between rendering and occlusion query state changes. In scenes with very high depth complexity, the occlusion queries dominate and with very low depth complexity the rendering

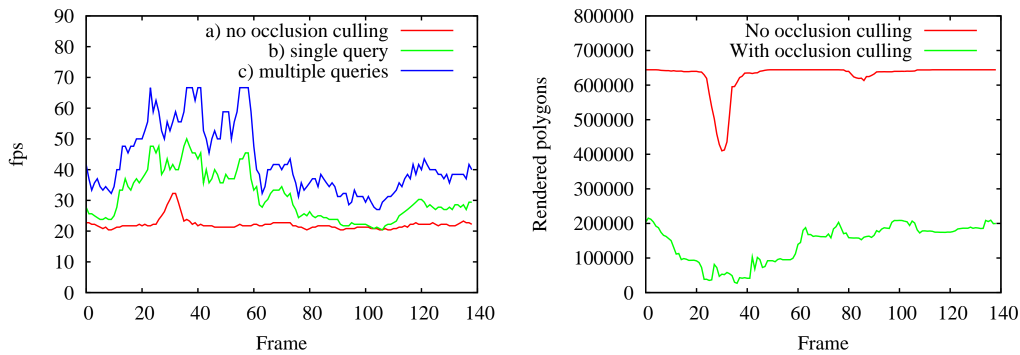

Figure 6.10: Boom Box frame rates and rendered polygons.

	Average Frame Rates [fps]		
	a) only vfc	b) synchronous occlusion queries	c) asynchronous multiple queries
Boom Box	22.1	31.1 +41%	42.9 +94%
Formula One Car	27.7	33.7 +22%	41.9 +51%
F1 Animation	8.3	11.1 +34%	16.5 +97%
Cotton Picker	8.8	17.9 +103%	20.6 +135%
Big City	0.5	19.9 +3880%	20.7 +4040%

Table 6.2: Resulting average frame rates and corresponding speed-ups for the test models.

dominates. The F1 Animation shows the lowest average frame rates, even though it has not the highest complexity, because only a few polygons were outside the view frustum during the camera path in contrast to the camera paths of the other models.

In Figures 6.10 – 6.11 the frame- and renderrates (how many polygons are rendered) are given. In scenes with higher depth complexity, the speed-up is not that high, because most of the costs are the fillrate for the occlusion tests and the scene graph traversal in the (occluded) backstage of the scene, which is similar with or without multiple queries.

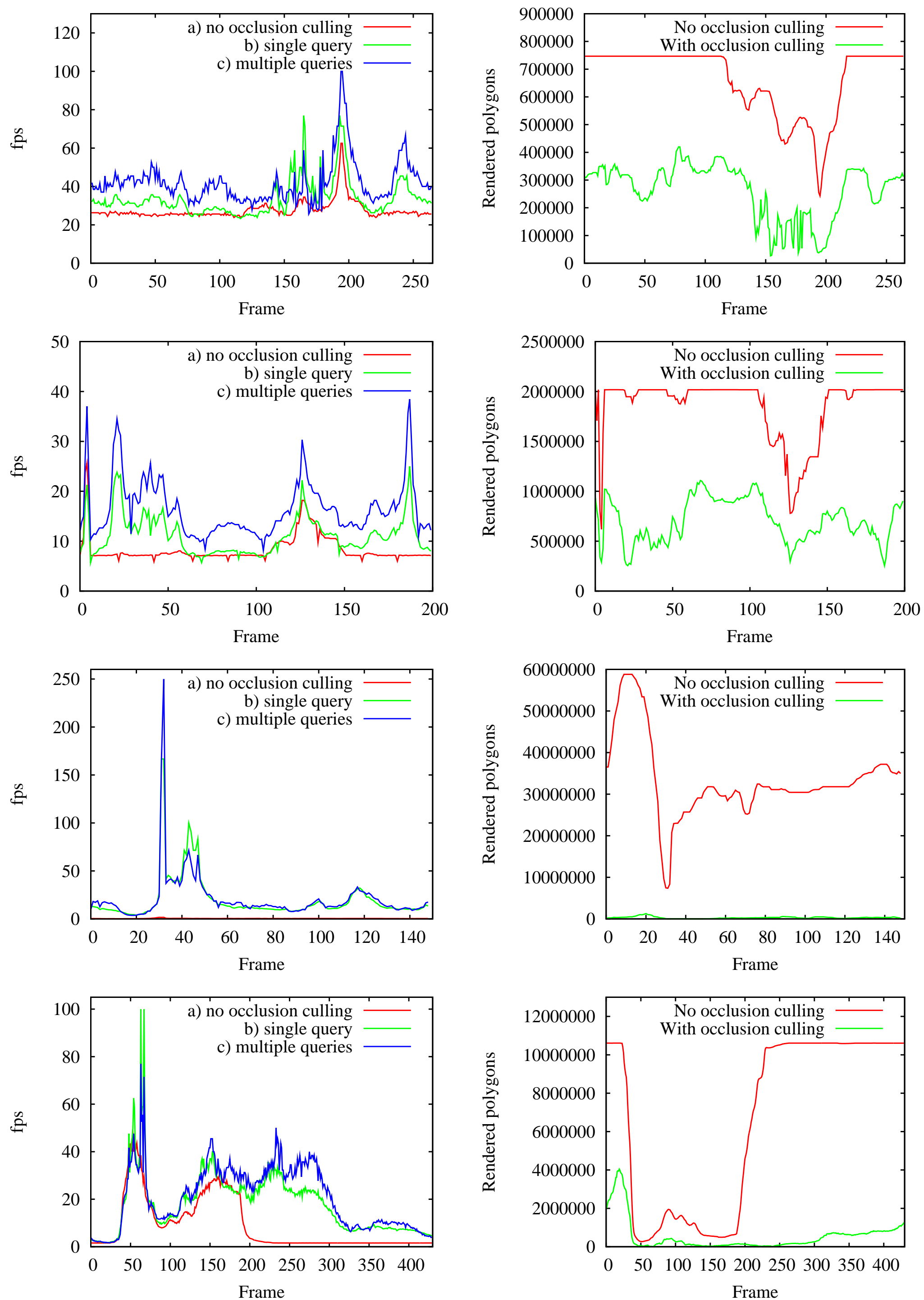

Figure 6.11: Frame rates (left) and rendered polygons (right) for the different models. Top-down: Formula One Car, F1 Animation, Big City, and Cotton Picker.

6.5.1 Scene Graph Structure

In the previous experiments, the original scene graphs of the models were used. But, obviously, the structure of the graph has an influence on the traversal and culling performance. Therefore a simple algorithm was used to generate different structures of the scene graphs. The algorithm arranges the geometry nodes in an rough octree structure with a predefined depth. Table 6.3 presents the average timings to generate an octree-like scene graph for the given models. The algorithm was not optimized, but shows that an update of the scene graph for each frame is too expensive, especially in large scenes.

Model	Approx. Time	Num. Geo. Nodes
Boom Box	315ms	530
Formula One Car	305ms	306
Cotton Picker	4 900ms	13 270
Big City	75 000ms	30 385

Table 6.3: Average time to generate an “octree-like” scene graph from the original graph.

In Table 6.4 and Figure 6.12 the average rendering performances for the different depths of the scene graphs are given. The results show that the performance is stable in scene graphs with a lower number of nodes. With a large number of nodes, a scene graph with a larger depth is needed to obtain good performance. In graphs with an almost flat structure and a large number of nodes, the needed sorting for the front-to-back rendering becomes expensive and limits the overall performance, because a huge number of nodes have to be sorted and traversed until the rendering can be started with the first geometry node. In scene graphs with larger depth, the sorting is executed parallel to the rendering (which is performed on the graphics hardware) and the culling can be hierarchically applied.

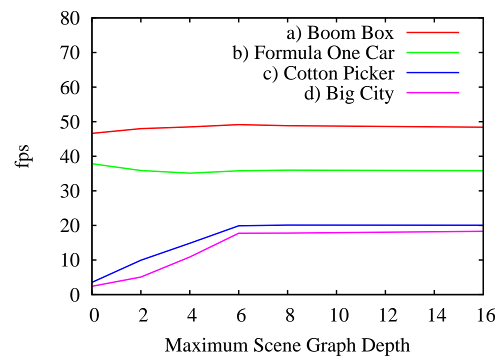

Figure 6.12: Average rendering performance depending on the used maximum scene graph depths.

Scene Graph Depth	Average Frame Rates [fps]						
	Original	Flat	2	4	6	8	Unlimited
Boom Box	42.8	46.6	48.0	48.5	49.2	48.9	48.5
Formula One Car	41.9	37.8	35.8	35.1	35.8	35.9	35.7
Cotton Picker	20.6	3.5	9.9	14.9	19.9	20.1	20.1
Big City	20.7	2.4	5.1	10.9	17.8	17.8	18.6

Table 6.4: Average rendering performance for different depths of the used scene graph.

Chapter 7

Conclusion and Future Work

7.1 Conclusions

In this thesis different occlusion culling techniques assisted by graphics hardware are evaluated. Additionally some new algorithms to reduce the number of hardware-assisted occlusion queries are presented. Also a novel algorithm for scene graph traversal to efficiently utilize multiple occlusion queries of the graphics hardware was introduced.

All the presented work can be used with arbitrary scenes and scene graph systems. Only a bounding volume hierarchy is needed to perform the occlusion queries. In the presented work we used only bounding boxes as bounding volumes, but also other bounding volumes can be utilized [1, 2].

The presented algorithms show good speed-ups compared to traditional rendering techniques without occlusion culling. Also they can be implemented on top of a wide range of different scene graph systems or other hierarchical data structures. No preprocessing on the data structures is needed, which makes the presented algorithms suitable for dynamic and interactive real-time environments.

All presented techniques can be combined with each other. This results in a very flexible solution for a wide range of different platform configurations. For example, a computer system with low graphics capabilities and without special hardware extensions can benefit from the presented occlusion query implementations. In contrast, a computer system with high performance graphics hardware and support for hardware occlusion queries will benefit from the special traversal technique to efficiently exploit the graphics hardware capabilities.

7.2 Main Results

Occlusion Query Implementations

Three different approaches to calculate the occlusion of a given bounding volume were compared in Chapter 4. All of them are using the depth values of the graphics hardware in any way to compare the depth values of the tested bounding volume with the depth values of already rendered geometry. The approaches differ in the utilization of the graphics hardware and the host's CPU. As expected, special extensions of the graphics hardware are the fastest way to get the occlusion. But also the other algorithms are able to speed up the rendering of complex scenes. Particularly they are useful in hardware environments without special occlusion query extensions. The read back of the z-buffer can be used in environments with high CPU power compared to the graphics capabilities, for example, in mobile phones or personal digital assistants (PDAs). On the other hand, the use of the OpenGL stencil buffer is useful in systems with middle-sized graphics hardware, missing the special OpenGL extensions, for example, in notebooks with unified memory architecture (UMA).

Avoiding Predictable Occlusion Queries

Because of the utilization of the graphics hardware for the occlusion queries, each occlusion query is associated with some latency. Therefore the number of occlusion queries to the graphics hardware has to be minimized to reduce the overall rendering time. We presented two novel techniques to decrease the number of occlusion queries in Chapter 5. The first technique focuses on occlusion queries with a visible result and the second on occlusion queries with an occluded result. Both techniques do not use the graphics hardware in some way to avoid additional latency. For example, with the Cotton Picker model, the number of occlusion queries decreased in average from 3456 to 1455 per frame for a given camera path. This is a reduction of 58%.

Occlusion Driven Traversal

Another way to reduce the latency for the occlusion queries is by using multiple occlusion queries in an interleaved fashion. Instead of synchronously waiting for an occlusion query result, this time is used for rendering or other occlusion queries. But the main problem is, how these multiple occlusion queries are organized. Which bounding volumes have to be added to a multiple occlusion query and which not? How can the bounding volumes be arranged to avoid false positive results? In Chapter 6 we presented an algorithm which uses screen-space coherence of the projected

bounding boxes to arrange the multiple occlusion queries during scene graph traversal. For example, the algorithm additionally gains over 5 fps in average for a F1 animation in a given camera path in contrast to synchronous occlusion queries, which is a speedup of 49%. Also the algorithm works in dynamic scenes and needs only an up-to-date bounding box hierarchy.

7.3 Future work

Hardware-assisted occlusion culling is a powerful tool to accelerate the rendering of very complex scenes with high depth complexity. The work presented in this thesis can be utilized in a standard scene graph environment. There are several directions and applications for future work:

- Beside the use for occlusion culling, a lower resolution bounding box of the Occupancy Map lookup gives information of the screen-space size for a node's bounding box. This can be used for level-of-detail selection or contribution culling.
- No temporal coherence is utilized in the presented approaches, but could be used to further speed up the construction and organization of the traversal and culling techniques. But this is a very sensitive technique and should be carefully used to avoid problems with dynamic scenes. This could also include a combination of the algorithm from Bittner et al. [16] and the presented traversal algorithm. For example, the selected nodes from the temporal coherence algorithm could be used as starting point for the occlusion-driven traversal. In frames with slow camera movement and less dynamic modifications, the algorithm would benefit from the temporal coherence. In dynamic, fast changing situations, the occlusion-driven part avoids loosing efficiency of the occlusion queries.
- Another obvious improvement is to render the geometry nodes in a state sorted fashion to further reduce the number of state changes during rendering. This can be easily implemented in the presented occlusion-driven traversal scheme.
- Precomputing was not in the focus, because dynamic scenes without assumptions on the scene graph have to work. In further releases this could become a more interesting point to speed up rendering of static or special scenes and could be implemented as additional tool complementing the presented techniques.

- The tests are working in a serial fashion, but could be parallelized, so that the software techniques, like the Occupancy Map or the Software Depth Buffer are working parallel to the hardware-assisted occlusion queries. This would result in a better load balancing between the main processor and the graphics subsystem and thus, to higher frame rates.

- All the presented work was done with a very flexible scene graph environment. If other spatial data structures are available, like an octree or BSP tree, several optimizations could be done. For example, the projection of the bounding boxes would not be necessary in an octree environment for the occlusion-driven traversal. The sorting in screen-space for the multiple occlusion queries can be implicitly done by a clever traversal of the octree.

Chapter 8

Appendix

8.1 OSGViewer

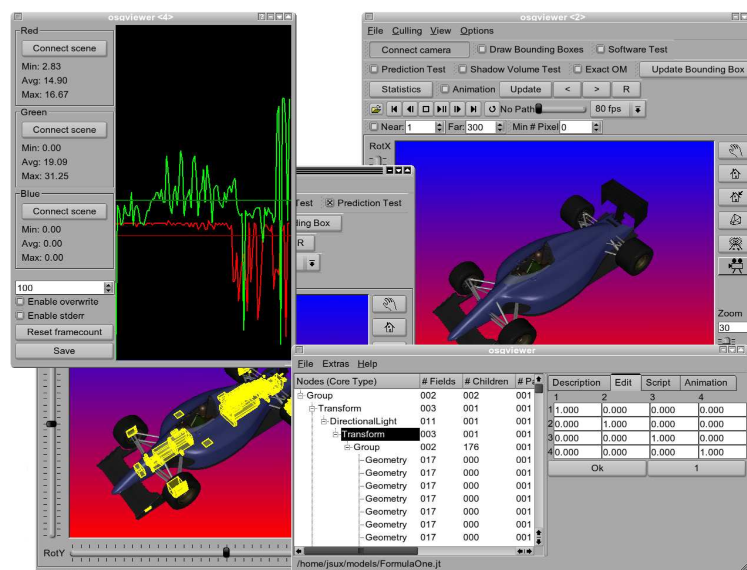

Figure 8.1: Screen shot of the OSGViewer.

The OSGViewer application was implemented to test the presented occlusion culling techniques with the OpenSG scene graph. In addition to the rendering capabilities, the application supports some features to manipulate a given scene graph, to manage animations, and to allow writing scripts. Also different file formats like the Jupiter-format are supported to import the models available from the Jupiter project.

8.1.1 Implementation

The application is written in C++ and uses Linux as platform. OpenSG is used as rendering and scene graph backend, Qt for the GUI and the QGLViewer for camera control. The SWIG library is used to attach different script languages to the C++ classes.

The application is split into two parts. The first part includes the “Main Window”, the I/O-capabilities and manages the central scene graph which contains the scene. The second part is responsible for the rendering of the scene graph in a “Scene Viewer”. Several instances of the Scene Viewer can be independently used at the same time.

8.1.2 Main Part

The main part controls the central scene graph and all functions, which do inspection or modifications of the central scene graph. Also multiple instances of the scene views are controlled. The main part is split into loading and saving, scripting and modification tools. The window in Figure 8.2 is used as user interface, which shows the scene graph as tree view in the left of the window. The user can inspect or manipulate the graph with the interfaces on the right side and with context-sensitive popup menus in the tree view.

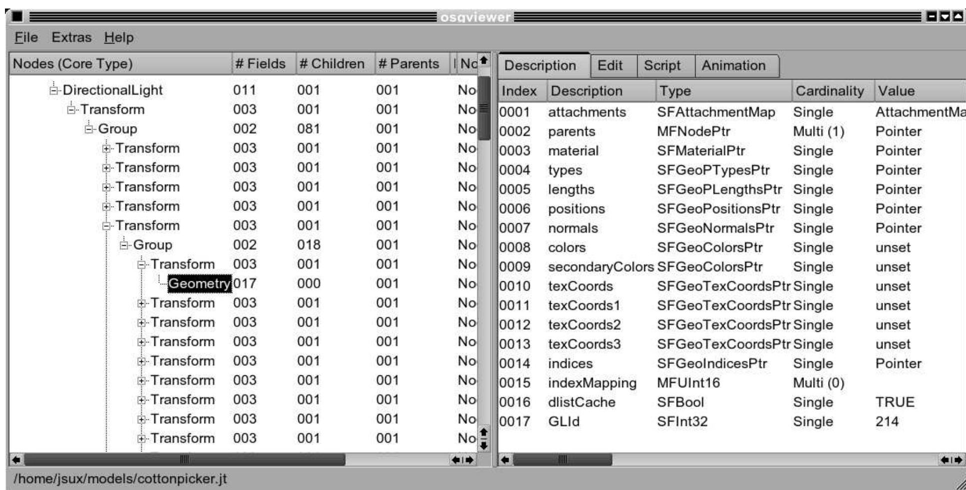

Figure 8.2: Main window of the OSGViewer.

For loading and saving of models, the applications supports different formats. This includes all OpenSG supported formats (VRML, OBJ and the own binary or

ASCII format). To benefit from the efficient binary format of the Jupiter scene graph, a special converter was implemented, which is able to convert an OpenSG scene graph to a Jupiter graph and vice versa. Also only subgraphs can be saved or added to an existing scene graph, which allows a simple construction of new models.

For further modifications of the scene graph, the user is able to add new nodes to the scene graph. Also cut and paste of a subgraph is possible. Additional tools allow to replicate a subgraph in different directions or can restructure the nodes in an octree like hierarchy.

Because of the reflection system of OpenSG the application can give the user full access to do modifications of the scene graph data like material or light parameters. Also some additional special interfaces are available for a fast editing of transformation matrices.

8.1.3 Scene View

In contrast to the abstract view of the scene graph in the main window, the scene view presents the 3d OpenGL rendering of the graph. To render a scene, the scene viewer uses an instance of a new render action, which is derived from the original render action provided by OpenSG. The new render action implements different enhanced techniques like the ones presented in Chapter 6. The interface (see Figure 8.3) of the scene viewer allows to configure a wide range of parameters for the rendering for example, enabling occlusion culling with the hardware extensions or enabling the Occupancy Map.

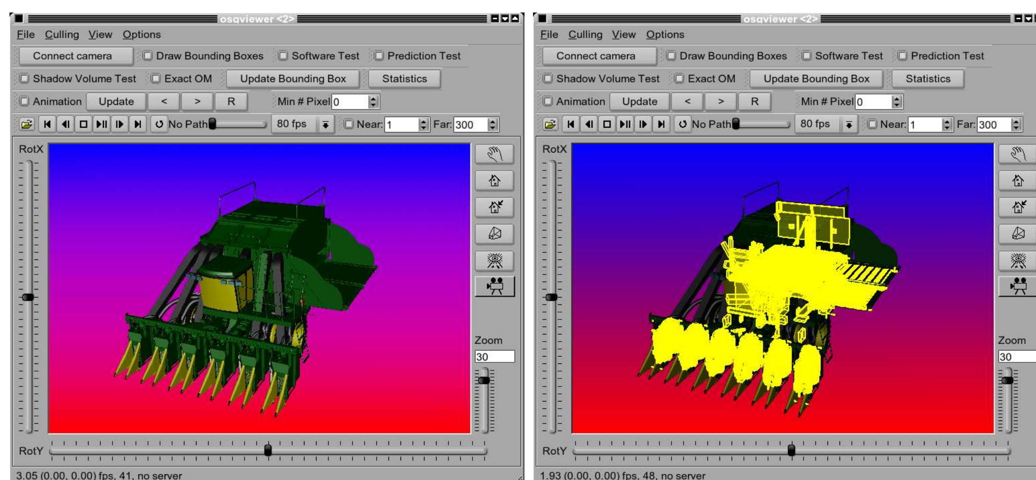

Figure 8.3: Scene view of the OSGViewer.

The *qglviewer* library was used as user interface to control the camera, which yields additional features like playing of camera paths or automatic spinning. Additionally, multiple scene views can be connected to each other. A “master” scene view controls the connected cameras. For example, this allows to compare the rendering speed of the same view at the same time for different techniques with and without occlusion culling.

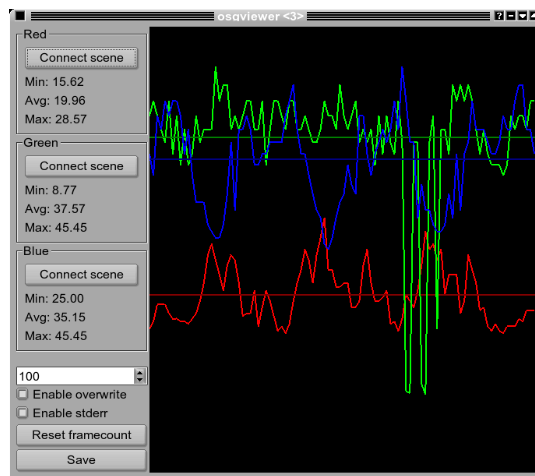

Figure 8.4: Screen shot of the statistics window.

To give a feedback of the rendering performance, the frame rate can be visualized in a statistics window. Figure 8.4 shows an example of this window. The user can connect a scene view with drag and drop to the statistics window. Different modes are available for the visualization.

8.2 Test Models

8.2.1 Cotton Picker

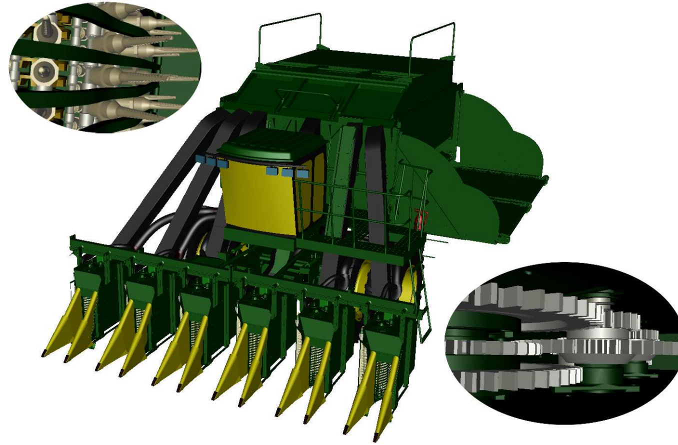

Figure 8.5: Complex scene of a Cotton Picker.

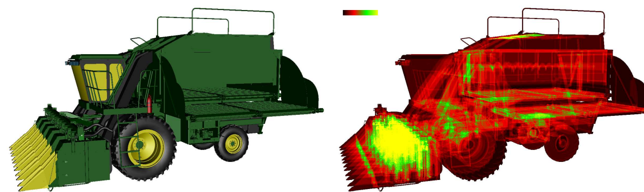

Figure 8.6: Depth complexity of the Cotton Picker model (avg. complexity: 13.3752 in 432 540 visible pixels).

Number of polygons	10 610 166
Number of nodes	40 724
Number of geometry nodes	13 270

Table 8.1: Scene specifications of the Cotton Picker model.

The Cotton Picker is a “real world” model from an industrial CAD modeling application. It consists of 13,270 individual parts in its assembly list (see Table 8.1). Most of the geometric complexity is located in the front of the model where are the spindle parts located. Each spindle has a huge number of small spikes. Depending

on the viewpoint most of the spindles are occluded by chassis parts. However, from a frontal point of view, most of the spindles are visible, which decreases the benefits of occlusion culling approaches.

8.2.2 Formula One Car

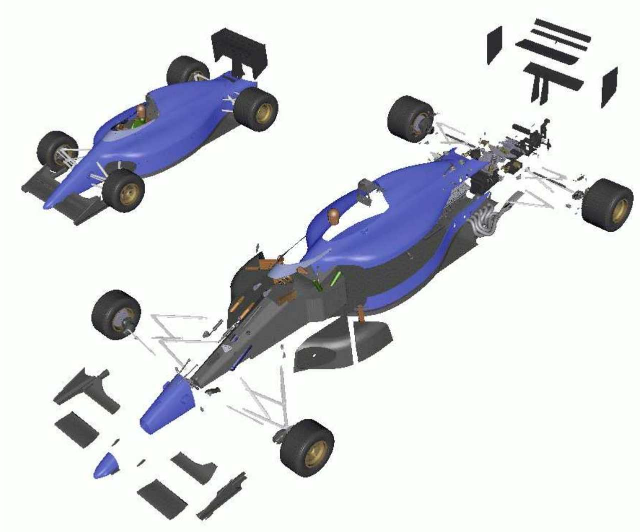

Figure 8.7: Formula One Car.

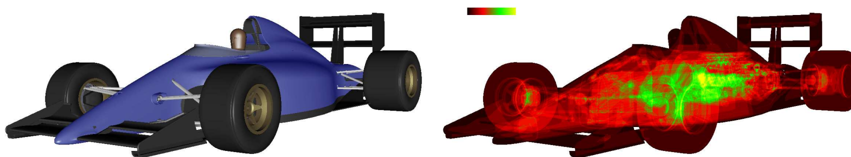

Figure 8.8: Depth complexity of the Formula One Car model.

Number of polygons	746 827
Number of nodes	311
Number of geometry nodes	306

Table 8.2: Scene specifications of the Formula One Car model.

The Formula One Car model is like the Cotton Picker a CAD model. It has a lower complexity with 746,827 polygons. Inside the chassis the model consists of

an engine, a gear box and suspension parts. Also a driver is sitting on the seat and controls the drive. The polygons are distributed over the whole body and chassis. In most points of view, the inner parts are occluded by the chassis.

F1 Animation

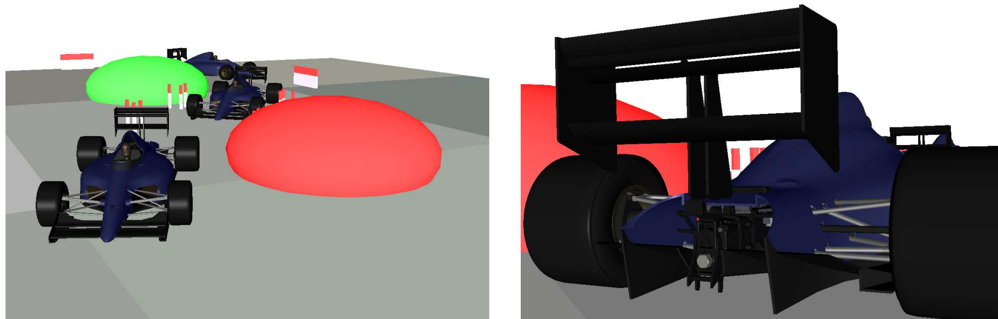

Figure 8.9: Screen shots of the F1 Animation model.

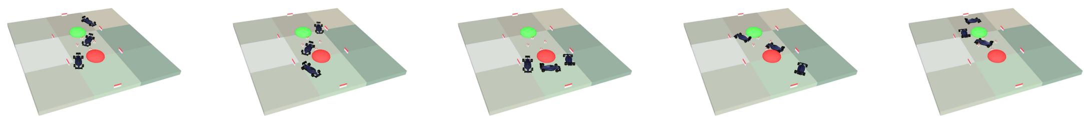

Figure 8.10: Some frames of the F1 Animation.

Number of polygons	2 242 481
Number of nodes	2 552
Number of geometry nodes	965

Table 8.3: Scene specifications of the F1 Animation model.

The F1 Animation is an artificial scene with three Formula One Car models and a very simple circuit in which the Formula One Car models describe an eight. Each car has a little different track and performance, however they do not overtake each other. The model is more complex with 2,242,481 polygons and is used to make experiments with dynamic scenes. The relations of the polygons to each other is changing in every frame, which has to taken into account in an occlusion culling approach.

8.2.3 City

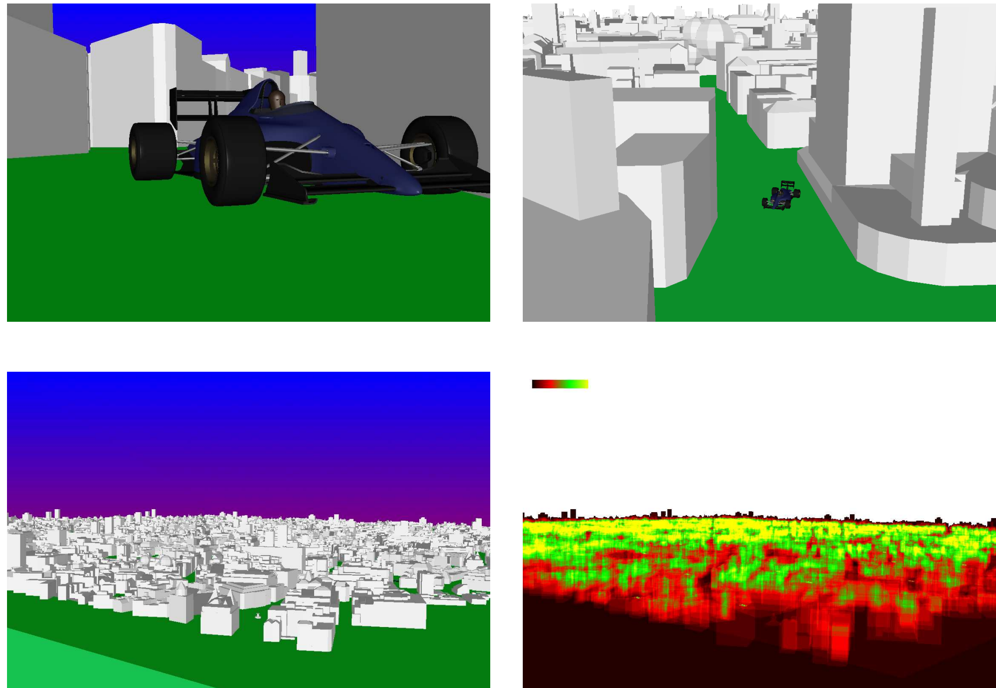

Figure 8.11: Different views of the Big City model, lower right: depth complexity.

Model	City	Big City
Number of polygons	4 056 195	64 898 464
Number of nodes	1 973	31 439
Number of geometry nodes	1 900	30 385

Table 8.4: Scene specifications of the both City models.

The City model is a combination of a London district and some Formula One Car models to increase the complexity. To get a larger model with higher complexity, the City model is replicated four times in each direction to build the Big City model. Both models are architectural scenes, which usually benefit from occlusion culling techniques. Facades in the front of the models occlude most of the inner and rear geometry.

8.2.4 Boom Box

The Boom Box model is a CAD model of a portable music system with lower complexity. In the interior the model contains some resistors, capacitors and other electronic stuff, which are occluded by the case.

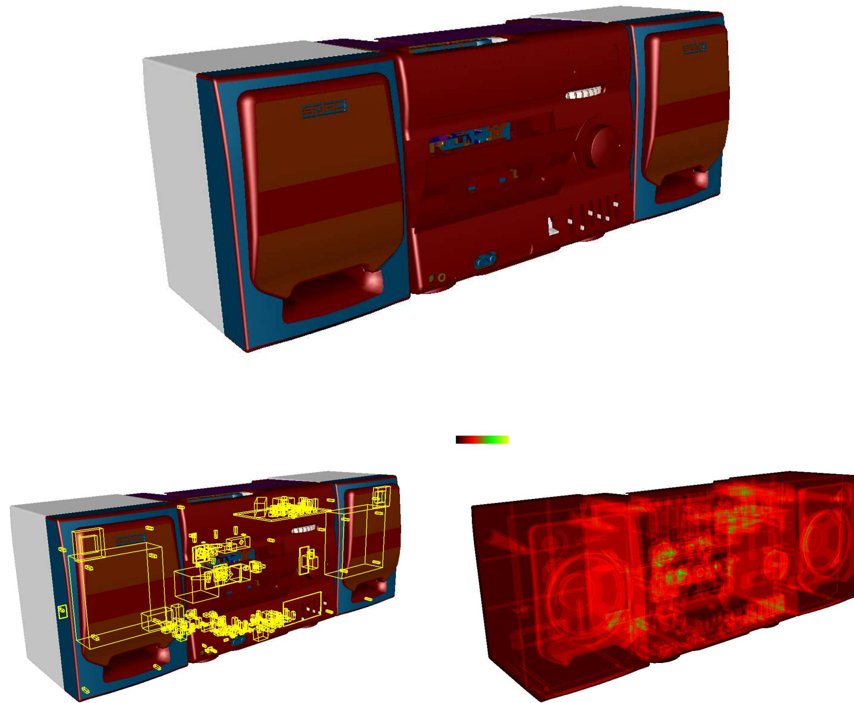

Figure 8.12: Image of the Boom Box model, lower left: bounding boxes of the interior, lower right: depth complexity.

Number of polygons	644 268
Number of nodes	1 277
Number of geometry nodes	530

Table 8.5: Scene specifications of the Boom Box model.

Bibliography

Author's list of publications

[1] D. Bartz, J. Klosowski, and D. Staneker. Tighter Bounding Volumes for Better Occlusion Performance. In *Visual Proc. of ACM SIGGRAPH*, page 213, 2001.

[2] D. Bartz, D. Staneker, and J. Klosowski. Tighter Bounding Volumes for Better Occlusion Culling Performance. Technical Report WSI-2005-13, Wilhelm Schickard Institute for Computer Science, Graphical Interactive Systems (WSI/GRIS), University of Tübingen, 2005.

[3] D. Bartz, D. Staneker, W. Straßer, B. Cripe, T. Gaskins, K. Orton, M. Carter, A. Johannsen, and J. Trom. Jupiter: A Toolkit for Interactive Large Model Visualization. In *Proc. of Symposium on Parallel and Large Data Visualization and Graphics*, pages 129–134, 2001.

[4] P. Biber, S. Fleck, M. Wand, D. Staneker, and W. Straßer. First Experiences with a Mobile Platform for Flexible 3D Model Acquisotion in Indoor and Outdoor Environments - The Wägele. In *Proc. of 3D-Arch*, 2005.

[5] D. Staneker. Ein hybrider Ansatz zur effizienten Verdeckungsrechnung. Master's thesis, University of Tübingen, Computer Science, 2001.

[6] D. Staneker. A First Step towards Occlusion Culling in OpenSG PLUS. In *Proc. of OpenSG Symposium*, 2002.

[7] D. Staneker. An Occlusion Culling Toolkit for OpenSG PLUS. In *Proc. of OpenSG Symposium*, 2003.

[8] D. Staneker, D. Bartz, and M. Meißner. Using Occupancy Maps for Better Occlusion Query Efficiency. In *EG Workshop on Rendering*, 2002. Poster.

[9] D. Staneker, D. Bartz, and M. Meißner. Improving Occlusion Query Efficiency with Occupancy Maps. In *Proc. of Symposium on Parallel and Large Data Visualization and Graphics*, pages 111–118, 2003.

[10] D. Staneker, D. Bartz, and W. Straßer. Efficient Multiple Occlusion Queries for Scene Graph Systems. Technical Report WSI-2004-6, Wilhelm Schickard Institute for Computer Science, Graphical Interactive Systems (WSI/GRIS), University of Tübingen, 2004.

[11] D. Staneker, D. Bartz, and W. Straßer. Occlusion Culling in OpenSG PLUS. *Computers & Graphics*, 28(1):87–92, 2004.

[12] D. Staneker, D. Bartz, and W. Straßer. Occlusion-Driven Scene Sorting for Efficient Culling. In *Proc. of Afrigraph, to appear*, 2006.

[13] G. Wetekam, D. Staneker, U. Kanus, and M. Wand. A Hardware Architecture for Multi-Resolution Volume Rendering. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 2005.

Literature

[14] D. Bartz, M. Meißner, and T. Hüttner. Extending Graphics Hardware for Occlusion Queries in OpenGL. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 97–104,158, 1998.

[15] D. Bartz, M. Meißner, and T. Hüttner. OpenGL-assisted Occlusion Culling of Large Polygonal Models. *Computers & Graphics*, 23(5):667–679, 1999.

[16] J. Bittner, M. Wimmer, H. Piringer, and W. Purgathofer. Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. In *Eurographics*, 2004.

[17] E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, 1974.

[18] D. Cohen-Or, Y. Chrysanthou, F. Durand, and C. Silva. Visibility: Problems, Techniques, and Application. In *ACM SIGGRAPH Course 4*, 2000.

[19] D. Cohen-Or, Y. L. Chrysanthou, C. T. Silva, and F. Durand. A Survey of Visibility for Walkthrough Applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3), 2003.

[20] OpenSG PLUS Consortium. OpenSG PLUS Project. `http://www.opensg.org/OpenSGPLUS/index.EN.html`, 2001.

[21] B. Cripe and T. Gaskins. The DirectModel Toolkit: Meeting the 3D Graphics Needs of Technical Applications. *The Hewlett-Packard Journal*, May:19–27, 1998.

[22] M. DeLoura. *Game Programming Gems 2 (T. Forsyth: Impostors: Adding Clutter)*. Charles River Media, 2001.

[23] R. Fernando and M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison–Wesley Professional, 2003.

[24] J. Foley, A. Van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice*. Addison–Wesley, 2nd edition, 1996.

[25] OpenSG Forum. OpenSG - Open Source Scenegraph. `http://www.opensg.org`, 2000.

[26] M. Garland. Multiresolution Modeling: Survey and Future Opportunities. In *Eurographics STAR report 2*, 1999.

[27] N. Greene, M. Kass, and G. Miller. Hierarchical Z-Buffer Visibility. In *Proc. of ACM SIGGRAPH*, pages 231–238, 1993.

[28] S. Guthe, M. Wand, J. Gonser, and W. Straßer. Interactive rendering of large volume data sets. In *IEEE Visualization*, 2002.

[29] P. Heckbert and M. Garland. Multiresolution Modeling for Fast Rendering. In *Graphics Interface*, 1994.

[30] Hewlett-Packard. Starbase Graphics Techniques. Volume 1, 1st edition, 1991.

[31] Hewlett-Packard. Occlusion Test, Preliminary. `http://www.opengl.org/Developers/Documentation/Version1.2/HPspecs/occlusion_test.txt`, 1997.

[32] Hewlett-Packard. Jupiter 1.0 Specification. Technical report, Hewlett Packard Company, Corvallis, OR, 1998.

[33] Hewlett-Packard. HP Visibility Test. `http://dune.mcs.kent.edu/~farrell/distcomp/graphics/hpopengl/Reference/glVisibilityBufferHP.html`, 1999.

[34] Silicon Graphics Inc. Cosmo3D Programmer's Guide. Technical report, 1998.

[35] W. Straßer J. Encarnação and R. Klein. *Graphische Datenverarbeitung I*. R. Oldenbourg Verlag, 1996.

[36] W. Straßer J. Encarnação and R. Klein. *Graphische Datenverarbeitung II*. R. Oldenbourg Verlag, 1997.

[37] F. Kahlesz, A. Balazs, and R. Klein. NURBS rendering in OpenSG PLUS. `http://www.opensg.org/OpenSGPLUS/symposium/`, 2002.

[38] J. Klosowski and C. Silva. The Prioritized-Layered Projection Algorithm for Visible Set Estimation. *IEEE Transactions on Visualization and Computer Graphics*, 6(2), 2000.

[39] J. Klosowski and C. Silva. Efficient Conservative Visibility Culling Using the Prioritized-Layered Projection Algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 7(4), 2001.

[40] T. Leyvand, O. Sorkine, and D. Cohen-Or. Ray Space Factorization for From-Region Visibility. In *Proc. of ACM SIGGRAPH*, 2003.

[41] D. Luebke and C. Georges. Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets. In *Proc. of ACM Symposium on Interactive 3D Graphics*, pages 105–106, 1995.

[42] M. Meißner, D. Bartz, R. Günther, and W. Straßer. Visibility Driven Rasterization. *Computer Graphics Forum*, 20(4):283–294, 2001.

[43] S. Nirenstein, E. Blake, and J. Gain. Exact from-region visibility culling. In *Proceedings of the 13th workshop on Rendering*, pages 191–202. Eurographics Association, 2002.

[44] NVidia. NVidia Occlusion Query. `http://oss.sgi.com/projects/ogl-sample/registry/NV/occlusion_query.txt`, 2001.

[45] D. Reiners. A Flexible and Extensible Traversal Framework for Scenegraph Systems. `http://www.opensg.org/OpenSGPLUS/symposium/`, 2002.

[46] J. Rohlf and J. Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. In *Proc. of ACM SIGGRAPH*, pages 381–394, 1994.

[47] R. J. Rost. *OpenGL Shading Language*. Addison–Wesley Professional, 2004.

[48] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison–Wesley, 1994.

[49] N. Scott, D. Olsen, and E. Gannett. An Overview of the VISUALIZE fx Graphics Accelerator Hardware. *The Hewlett-Packard Journal*, (May):28–34, 1998.

[50] M. Segal and K. Akeley. The OpenGL Graphics System: A Specification (2.0). `http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf`, 2004.

[51] L. A. Shirman and S. S. Abi-Ezzi. The Cone of Normals Technique for Fast Processing of Curved Patches. In *Eurographics*, pages 261–272, 1993.

[52] P. Slusallek, S. Parker, E. Reinhard, H. Pfister, and T. Purcell. Interactive Ray-Tracing. In *ACM SIGGRAPH Course 13*, 2001.

[53] M. Stein, E. Bowman, and G. Pierce. *Direct3D: Professional Reference*. New Riders Pub, 1997.

[54] W. Straßer. *Schnelle Kurven- und Flächendarstellung auf graphischen Sichtgeräten*. PhD thesis, Technische Universität Berlin, 1974.

[55] G. Voß, J. Behr, D. Reiners, and M. Roth. A multi-thread safe foundation for scenegraphs and its extension to clusters. In *EG Workshop on Parallel Graphics and Visualisation*, 2002.

[56] M. Wand. Point-based multi-resolution rendering. Dissertation, University of Tübingen, 2004.

[57] M. Wand, M. Fischer, I. Peter, F. Meyer auf der Heide, and W. Straßer. The randomized z-buffer algorithm: Interactive rendering of highly complex scenes. In *SIGGRAPH 2001*, 2001.

[58] M. Weiler and T. Ertl. Ein Volume-Rendering-Framework für OpenSG. `http://www.opensg.org/OpenSGPLUS/symposium/`, 2002.

[59] J. Wernecke. *The Inventor Mentor*. Addison–Wesley, 1994.

[60] P. Wonka, M. Wimmer, and D. Schmalstieg. Occluder Shadows for Fast Walkthroughs of Urban Environments. In *Computer Graphics Forum*, pages vol. 18, no. 3, pp. 51–60, 1999.

[61] M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide*. Addison–Wesley, 2nd edition, 1997.

[62] WSI/GRIS. The Kelvin Project. `http://www.gris.uni-tuebingen.de/kelvin/`, 1999.

[63] H. Zhang, D. Manocha, T. Hudson, and Kenneth E. Hoff. Visibility Culling Using Hierarchical Occlusion Maps. In *Proc. of ACM SIGGRAPH*, pages 77–88, 1997.